Machines and Their Languages
G51MAL

Dick Crouch

Semester 2, 1997/98

Contents

Chapter 1

Introduction

1.1 What this course is about

Applications of Formal Language Theory

The manipulation and processing of formal languages is a central part of computer science. Programming languages, like C++ or Java, are formal languages. Compilation of these languages into machine code relies heavily on concepts and techniques developed in formal language and automata theory.

Work in computational linguistics and natural language processing has also borrowed heavily from formal language theory. Although there is room for doubt about whether natural languages like English are formal (in the sense of the term that will shortly be defined), many techniques can usefully be adopted. Indeed, some of the theory was developed by linguists rather than mathematicians or computer scientists.

At a more abstract level, one can view computers as devices for manipulating expressions in a language whose sentences consist of sequences of zeroes and ones. In this setting, formal language theory can be used to establish some theoretical limits on what can and cannot be done with computers.

Warnings

Although formal language and automata theory has a range of practical applications, this course aims primarily at setting out the underlying theory. Because of this, a few health warnings are advisable.

First, this course will not teach you how to write better programs or explain how computers work. It will not enable you to write fancy graphical interfaces or web applications to amaze and impress your friends. Some software tools will be made available to demonstrate certain aspects of the subject, but the course is theoretical rather than practical.

Second, the subject matter is unavoidably mathematical. The level of mathe-

matics required is not that deep: the first year courses on maths for computer science will provide you with all you need to know. However, the mathematics crops up all the time — what it lacks in complexity it makes up for in quantity.

As far as possible I will avoid unnecessary mathematical clutter, and separate descriptions of concepts and techniques from their mathematical justifications. However, you will be expected to look at formal definitions, proofs and lemmas and be able to make intuitive sense of them. This is a skill that has to be learned, and is a pre-requisite for picking up any text-book on the subject to supplement the lectures and notes.

Third, this is a subject that is hard to learn passively: you will need to do lots of exercises if you want to do well in the exams. There will be weekly problem classes in addition to the lectures, and attendance while optional is strongly advised.

You will probably find that the course introduces you to plethora of new terms and concepts. Beneath this variety there are a number of basic unifying themes. The subject is a fundamentally quite simple one. However, the simplicity is not immediately apparent: you need to put some hard work in before things start to drop into place. And if you don't put the initial work in, the subject will remain bewildering.

If this list of warnings has not completely put you off, read on ...

Reading List

There are a number of textbooks covering most of the material in this course. You will need to consult at least one of these as an adjunct to the notes and the lectures. Some of these texts, in a rough preference order (best first) are:

1. D. Kelley, 1995, *Automata and Formal Languages*, Prentice Hall

   Compact, but with lots of examples and exercises. Does not cover everything in the course, e.g. nothing on parsing, but includes most of what you need to know

2. T.A. Sudkamp, 1997, *Languages and Machines*, Addison Wesley.

   Longer than Kelley, but covers everything in the lectures and more. Also contains lots of examples and exercises

3. V.J. Rayward-Smith, 1983, *A First Course in Formal Language Theory*, Blackwell.

   Short and relatively accessible, but rather compactly laid out and does not cover Turing machines.

4. D.I.A. Cohen, 1997, *Introduction to Computer Theory* (2nd edition), John Wiley.

   Chatty and relatively non-mathematical (hence, too long)

5. J.E. Hopcroft & J.D. Ullman, 1979, *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley.

Closer to a standard reference than an introduction: heavy going

1.2 Some Fundamental Concepts

1.2.1 Formal Languages

What is a formal language? Here are some technical definitions:

1. A **language** is a set of **sentences**
2. A **sentence** is an ordered sequence of **words**, taken from a specified (and finite) **vocabulary**.
3. A **vocabulary** is a finite set of symbols, where the symbols are known as words.

   (N.b. some texts use the term *alphabet* instead of *vocabulary*, *letter* instead of *word*, and *word* instead of *sentence*.)
4. A **string** is any ordered sequence of words taken from a specified vocabulary.

   Thus all the sentences of a language are strings over the vocabulary of the language. But usually, not all possible strings are sentences of the language
5. A **formal language** is a language for which there is a precise and finitely specifiable set of rules for identifying all and only the sentences of the language

Before giving examples, note that these are narrow, technical definitions of languages and sentences, and are not intended to capture all of what we intuitively understand by these terms.[1]

Consider the English language. This has a finite (though large) vocabulary of individual words. These words can be put together in the right order to form sentences belonging to the English language, e.g

The cat sat on the mat

They can also be put together in the wrong order to produce strings of words that are not English sentences and do not belong to the language, e.g.

The on cat mat sat

The cat sat on the

From a finite vocabulary, English generates an infinite set of sentences. Infinitely large languages are the rule rather than the exception, though finite languages do also exist.

[1] For example, the definitions so far say nothing about what sentences mean, or how they might be used for communication or other purposes.

It is unclear whether English is a formal language. There are some sequences of words where it is unclear if they are proper English sentences or not. And even if precise decisions can be made in all cases, no one has yet succeeded in stating a finite and precise set of rules for mechanically deciding every issue.

Programming languages like C++ definitely are formal languages. Anyone who has ever struggled with a C++ compiler will know that there are quite rigid rules about what do and do not count as valid programs. It is the existence of these rules that makes the language a formal one. The role of sentences in C++ is played by complete programs, and the language is infinite since there are an infinite number of different programs.

Another example of a formal language is the language of arithmetic, and we will look at this one in more detail:

1.2.2 The Language of Arithmetic

First, some examples of sentences in the language of arithmetic

$$2 + 2 = 4$$
$$2 + 2 = 5$$
$$2 + (5 * 8)/4 = 22$$

and a non-sentence

$$+ * 2 =)2(1$$

The vocabulary of this language is the set of symbols

$$\text{Vocabulary} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, *, /, =,), (, \Delta\}$$

(Note that Δ is used to explicitly represent white space. Although in written English there is a convention that white space separates individual words, in other languages the space may be a word in itself).

We can form numbers as sequences of words. The number 23 is the word 2 followed by the word 3, with no intervening space.

The sentences of the language are two arithmetic expressions (expressions that evaluate to numbers) separated by the equals sign, =. Note that the sentences of the language do not have to be true. Thus $2 + 2 = 5$ is a sentence of the language; unlike $2 + 2 = 4$, it happens to be a false sentence.

It should be obvious that the language of arithmetic comprises an infinite number of sentences. But not all strings of words are sentences of the language.

Within the language of arithmetic, we can identify a more specialised sub-language: the language of (integer) numbers. The vocabulary of this language is $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, and it is the numbers themselves that are the sentences of this language. (Note that numbers are *not* sentences in the language of

arithmetic, and so the two languages are non-overlapping even though there is an overlap in the vocabulary)

The language of arithmetic, and the language of numbers, is formal because precise rules can be given to determine which strings of words are and are not sentences of the language. We now turn to how these rules are stated.

1.2.3 Rules for Formal Languages

There are two main ways of giving rules for determining whether a string of words is a sentence of a formal language or not. One is by specifying a **grammar** for the language. The other is by specifying an **automaton** (or machine) for the language. One of the main results of formal language and automata theory is establishing the equivalence between different classes of grammar on the one hand and automata on the other.

But before turning to grammars and automata, we should first consider a less widely applicable way of defining membership of a language. Recall that a language is simply a set of sentences. We might therefore write out all the sentences in the language, and check whether or not a given string is a member of that set. But this will only do for finite languages. Recall that the rules defining a formal language must be finitely specifiable. We cannot give a finite specification by writing down an infinite set of sentences.

This is why grammars and automata are so useful. They give a finitely specifiable way of generating an infinite set of sentences.

1.2.4 Grammars

A grammar consists of a finite set of rules, each having the form

⟨left hand side⟩ → ⟨right hand side⟩

The following is a grammar for defining the language of binary numbers:

1. Bin → 0
2. Bin → 1
3. Bin → 1 ZBin
4. ZBin → 0
5. ZBin → 1
6. ZBin → 0 ZBin
7. ZBin → 1 ZBin

The rules say that a binary number, Bin, can be either a zero (rule 1), a one (rule 2), or (rule 3) a one followed another binary number ZBin (which may have leading zeroes). A binary number with leading zeroes, ZBin, may be a zero (rule 4), a one (rule 5), or a zero or one followed by another number with leading zeroes (rules 6 & 7).

The arrow $\rightarrow$ in the rules says that the symbol on the left hand side of the rule may be rewritten, or replaced by the symbols on the right hand side. The symbols 0 and 1 are the words of the language being defined. The symbols Bin and $ZBin$ are variables that can be replaced, through applying the rule, by sequences of words.

To show that 101 is a sentence in the language of binary numbers, we carry out a **derivation**. This involves first writing down the symbol Bin, and successively applying the grammar rules until we obtain the string we are looking for:

String	Rule
Bin	Bin $\rightarrow$ 1 ZBin
1ZBin	ZBin $\rightarrow$ 0 ZBin
10ZBin	ZBin $\rightarrow$ 1
101	Done

At each step, we pick a rule to replace one of the variables in the string by whatever occurs on the right hand side of the rule. The derivation stops when we have a string consisting entirely of words, to which no further rules can be applied.

Exercise Find a derivation for the number 1100. Satisfy yourself that there is no derivation for 010

Terminology For Grammars

Grammars operate with two disjoint sets of symbols

1. Words (also known as **terminal symbols**)

   These are the words that actually occur in the sentences of the language

2. **Variables** (also known as **non-terminal symbols**)

   These never occur in sentences of the language, and grammar rules are used to rewrite them to strings of words

Amongst the variable (non-terminal) symbols, there is one distinguished symbol known as the **start symbol**. Derivations of sentences in the language always begin by writing down a string consisting of a single occurrence of the start symbol, and then repeatedly applying rules until a string is obtained consisting purely of words (terminal symbols).

All grammar rules have the form

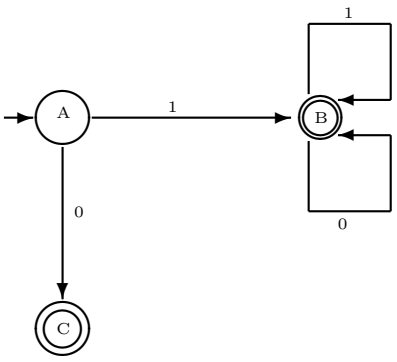
LHS-String → RHS-String

where the LHS and RHS are string of variables and words. Different classes of grammar emerge from applying different restrictions on what kinds of string can occur on the left and right hand sides of the rules. In the grammar above, the left hand string always consists of a single variable symbol (Bin or $ZBin$). The right hand string consists of a mixture of variables and words.

1.2.5 Automata

Automata, or state machines, provide an alternative way of specifying the rules defining formal languages. A state machine consists of a number of states connected by labelled arrows. These arrows denote ways of making transitions from one state to another.

Here is a state machine defining the language of binary numbers:



This consists of three states (labelled A, B and C) and four transitions. State A is the **start state**, as indicated by the unlabelled transition arrow coming into it. States B and C are **accepting states**, as indicated by the double circles. The transitions between states are labelled, in this case with the words 1 or 0.

To use the automaton to show that the string 1010 is a sentence of the language, begin in the start state. Look at the first word in the string, which is 1. Take the transition labelled with a 1 to the next state (i.e. state B). Look at the next word in the string, which is 0. Take the transition from state B labelled with a 0. This loops back to state B. Look at the next word (1) and take the 1-transition from B to B. Look at the next (and final) word, 0. Take the 0-transition from B to B. We have now got to the end of the string, and we are in an accepting state. This means that the string is a sentence in the language defined by the automaton.

To show that 011 is not a string in the language, again begin in the start state A. The first word is 0, so we have to take the 0-transition to state C. The next word is 1, but there are no 1-transitions leaving state C. In other words, we cannot read through to the end of the string to arrive in an accepting state, and so the string is not a sentence of the language.

The machine shown above, known as a **finite state automaton**, has no memory. It can keep no track of which states it has been through in the past. More powerful state machines can be formed by adding different levels of memory to the machine (e.g. a stack-like memory, a limited size random access memory, and unlimuited size random access memory). These different types of machine can be used to define different classes of language.

1.2.6 The Chomsky Hierarchy

There are different classes of grammars and different classes of automaton, defining different classes of language. In fact, there is a strong parallelism between classes of grammar and classes of automaton. For the classes of language we will be looking at in this course, there will be parallel grammars and automata defining them.

The different classes of grammar arise from imposing different restrictions on the strings of symbols that can occur on the left and right hand sides of rules. The different classes of machine are reflected by different types of labelling on the transitions, which indicate different types of memory associated with the machine. The following hierarchy emerges:

Language Type	Grammar Restriction	Automaton (& memory)
Regular	LHS single variable RHS either (a) a single word, or (b) a single word plus single variable	Finite state automata (No memory)
Context Free	LHS single variable RHS can be anything	Pushdown stack automata (Stack memory)
Context Sensitive	RHS no shorter than LHS	Linear bounded automata (Limited RAM)
Recursively Enumerable	No restrictions	Turing machines (Unlimited RAM)

The further up the hierarchy (towards recursively enumerable languages and Turing machines), the more control can be exercised over including and excluding different types of string in the language.

Regular languages, grammars and finite state machines can be very efficiently processed. They form the basis for a number of unix tools (e.g. regular expressions in grep). A lot of speech and natural language processing nowadays attempts to improve processing times by approximating natural languages to regular languages (unfortunately, this *is* an approximation — natural languages are at least context free).

Context free languages are very widespread. Most programming languages are context free, and natural languages are more or less context free. Techniques for defining and analysing sentences in context free languages form the basis of compiler theory.

Context sensitive languages are of comparatively little practical or theoretical interest (though it was once though that all natural languages must be at least context sensitive: this now seems unlikely).

Turing machines are of interest because they provide a simple but powerful way of modelling the functionality of digital computers. Turing machines can be used to establish a number of theoretical results about limits on what is computable. However, the languages they define tend not to be of much practical interest.

There are also other classes of language, grammar and automaton lying between these main points on the Chomsky hierarchy. Some of them will be mentioned in passing later on. The course is structured around an ascent through the hierarchy, starting with regular languages and finite state automata, and ending at Turing machines.

1.3 Mathematical Preliminaries

The following should all be familiar, so only brief notes will be given

1.3.1 Sets

Languages are sets of sentences. Operations on languages are therefore primarily set operations, of which the main ones are:

- $S = \{e_1, e_2, \ldots, e_n\}$
  $S = \{x \mid f(x)\}$
- Membership, $\in$
  $e_1 \in \{e_1, e_2, \ldots, e_n\}$
  $e \in \{x \mid f(x)\}$ if $f(e)$
- Subset, $\subset$
  $S1 \subset S2$ if every member of S1 is also a member of S2
- Union, $\cup$
  $S1 \cup S2 = \{x \mid x \in S1 \text{ or } x \in S2\}$
- Intersection, $\cap$
  $S1 \cap S2 = \{x \mid x \in S1 \text{ and } x \in S2\}$
- Complement, $\bar{}$
  $\overline{S1} = \{x \mid x \notin S1\}$

- Difference, $-$
  $S1 - S2 = \{x \mid x \in S1 \text{ but } x \notin S2\}$
- Powerset, $\mathcal{P}$
  $\mathcal{P}(S) = \{S' \mid S' \subseteq S\}$
- 'Big' Union, $\bigcup_{i=1}^{n}$
  $\bigcup_{i=1}^{n} S_i = S_1 \cup S_2 \cup \ldots \cup S_n$
- 'Big' Intersection, $\bigcap_{i=1}^{n}$
  $\bigcap_{i=1}^{n} S_i = S_1 \cap S_2 \cap \ldots \cap S_n$
- $\mathcal{U}$, the universe — the set of all elements
- $\{\}$, the empty set

Some important properties of these operations are

Associativity	$(A \cup B) \cup C = A \cup (B \cup C)$ $(A \cap B) \cap C = A \cap (B \cap C)$
Commutativity	$A \cup B = B \cup A$ $A \cap B = B \cap A$
Complement	$A \cup \overline{A} = \mathcal{U}$ $A \cap \overline{A} = \{\}$
Idempotency	$A \cup A = A$ $A \cap A = A$
Identity	$A \cup \{\} = A$ $A \cap \mathcal{U} = A$
Zero	$A \cup \mathcal{U} = \mathcal{U}$ $A \cap \{\} = \{\}$
Involution	$\overline{(\overline{A})} = A$
De Morgan	$\overline{(A \cup B)} = \overline{A} \cap \overline{B}$ $\overline{(A \cap B)} = \overline{A} \cup \overline{B}$
Distributivity	$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

Note also that the empty set is a subset of every set

1.3.2 Relations

Relations can be defined in terms of sets. In particular, sets of n-tuples (a 2-tuple is called an ordered pair). N-tuples are defined in terms of the Cartesian product of sets.

- **Cartesian product:**
  $A \times B = \{\langle a, b\rangle \mid a \in A, b \in B\}$

I.e. the set of all ordered pairs (or 2-tuples), $\langle a, b\rangle$ where a is from set A and b from set B.

Can also have n-tuples: $A \times B \times \ldots \times N$

Any **binary relation** R on the members two sets A and B is some subset of their Cartesian product:

$$R \subset A \times B$$

(n.b. A and B can be the same set)

That is, a binary relation is a set of ordered pairs For example:

A = people, B = newspapers, R = reads

$R = \{\langle \text{smith, independent}\rangle, \langle \text{smith, guardian}\rangle, \langle \text{jones, telegraph}\rangle\}$

R(smith,telegraph) iff $\langle \text{smith, telegraph}\rangle \in R$

N-ary (or N-place) relations are generalisations of binary relations — sets of n-tuples

If we consider a binary relation R on a single set S, $R \subseteq S \times S$, there are a number of important properties that the relation may or may not exhibit. As exemplars, consider the binary relations defined on the set of numbers: $\geq$, $>$,$\leq$, $<$ and $=$.

- R is **reflexive** iff
  $R(s_i, s_i)$ for all $s_i \in S$

  $\geq$, $\leq$ and $=$ are reflexive
  $<$ and $>$ are not

- R is **symmetric** iff
  $R(s_j, s_k)$ implies $R(s_k, s_j)$

  $=$ is symmetric
  $\geq$, $\leq$, $<$ and $>$ are not

- R is **transitive** iff
  $R(s_j, s_k)$ and $R(s_k, s_m)$ implies $R(s_j, s_m)$

  $\geq$, $>$,$\leq$, $<$ and $=$ are all transitive

- If a relation is transitive, reflexive and symmetric
  it is an **equivalence** relation (like $=$)

- Transitive/reflexive/symmetric **closures**:
  Add just as many new ordered pairs to the relation as are needed to make it transitive/reflexive/symmetric

  e.g. $R = \{\langle a, b\rangle, \langle b, c\rangle\}$

Transitive closure: $\{\langle a, b\rangle, \langle b, c\rangle, \langle a, c\rangle\}$
Reflexive closure: $\{\langle a, b\rangle, \langle b, c\rangle, \langle a, a\rangle, \langle b, b\rangle, \langle c, c\rangle\}$
Symmetric closure: $\{\langle a, b\rangle, \langle b, c\rangle, \langle b, a\rangle, \langle c, b\rangle\}$

1.3.3 Functions

Functions are a special type of relation. Consider the one-place function *age-in-years*, *Age*

$$Age\text{: } People \longrightarrow Numbers$$

This is a function from the set of people into the set of numbers. The one place function is a two-place relation

$$Age \subseteq People \times Numbers$$
e.g. $Age = \{\langle tom, 18\rangle, \langle dick, 35\rangle, \langle harry, 21\rangle\}$

Not any type of relation will do. Every person $p \in People$ must occur in exactly one ordered pair in *Age* i.e. each person is paired with exactly one age (though there may be several different people paired to the same age)

In *Age*, the set *People* is the **domain** of the function (the argument) and $Number$ the **range** (result)

More generally, an N-place function is an N+1-ary relation

$$Dom_1 \times \ldots \times Dom_n \longrightarrow Range$$

where Every n-tuple of arguments occurs in just one n+1-tuple of the relation.

A **total** n-place function is one which pairs every possible domain n-tuple with an element in the range (i.e. it returns a value for all possible arguments). A **partial** function may fail to return a value for some combination of arguments:

$$Dom_1 \times \ldots \times Dom_n \mapsto Range$$

Not every domain tuple of arguments occurs in the function. But those that do occur, occur just once

1.3.4 Cardinality and Countability

The size (**cardinality**) of a set is given by putting the members of the set into one-to-one correspondence with the set of natural numbers, $\{1, 2, 3, \ldots\}$.

Languages often comprise infinitely large sets of sentences. It turns out that not all infinite sets are the same size. Some can have their elements put into one-to-one correspondence with the natural numbers — **countably** or **denumerably** infinite sets. Others have more members than there are natural numbers — **uncountable** sets.

For example, the set of rational numbers (fractions) is countably infinite. There are as many fractions as there are whole numbers. This may seem surprising, since between any two integers there are a infinte number of fractions. Nevertheless, the fractions can all be counted. We can enumerate them in s specific order that doesn't miss any out, and ensure that for any given fraction we will reach it in the order within a finite amount of time.

Here is a way of enumerating the fractions that goes through at each rational number at least once. We count the numbers by following the arrows

	1		2		3		4	...
1	1/1	→	1/2		1/3	→	1/4	...
		↙		↗		↙		
2	2/1		2/2		2/3		2/4	...
	↓	↗		↙		↗		
3	3/1		3/2		3/3		3/4	...
		↙		↗				
4	4/1		4/2		4/3		4/4	...
	↓	↗						
⋮	⋮		⋮		⋮		⋮	

Since we can regard fractions as pairs of numbers, this shows that the set of all pairs of elements taken from a countably infinite set is also countably infinite. The same holds of n-tuples.

However, the set of real numbers is not countable — there are more real numbers than integers or fractions. The proof of this is due to Cantor. Suppose that we do have some enumeration of the real numbers: we can show that whatever the enumeration there will always be at least one number it leaves out. Let us just consider some enumeration of the reals between zero and one:

$$\begin{array}{ll} 1 & 0.n_{1,1}n_{1,2}n_{1,3}n_{1,4}\ldots \\ 2 & 0.n_{2,1}n_{2,2}n_{2,3}n_{2,4}\ldots \\ 3 & 0.n_{3,1}n_{3,2}n_{3,3}n_{3,4}\ldots \\ 4 & 0.n_{4,1}n_{4,2}n_{4,3}n_{4,4}\ldots \\ \vdots & \end{array}$$

where $n_{i,j}$ is a digit in the decimal representation.

From this enumeration we can generate a diagonal number:

$$0.n_{1,1}n_{2,2}n_{3,3}n_{4,4}\ldots n_{i,i}\ldots$$

Thus, if our enumeration is

$$\begin{array}{ll} 1 & 0.1567... \\ 2 & 0.5267... \\ 3 & 0.5637... \\ 4 & 0.5674... \\ \vdots & \end{array}$$

the diagonal number will be 0.1234...

Create a new number from the diagonal number by altering each digit $n_{i,i}$ as follows

- If $n_{i,i} < 9$, then $n'_{i,i} = n_{i,i} + 1$
- If $n_{i,i} = 9$, then $n'_{i,i} = 0$

For example, this turns the diagonal number above into 0.2345....

Whatever the enumeration we had, we can be sure that the number

$$0.n'_{1,1}n'_{2,2}n'_{3,3}n'_{4,4}\ldots n'_{i,i}\ldots$$

did not occur in it. For any position i in the enumeration, the ith number differs from the new number in at least the digit $n_{i,i}$.

This means that we cannot hope to enumerate all the real numbers, and hence we cannot pair them one-to-one with the natural numbers.

1.3.5 Recursion

Recursive definitions are a very useful way of specifying infinite sets. A recursive definition always has:

1. one (or more) base cases,
2. one (or more) recursive steps,
3. and an exclusion condition (often implicit)

Here is an example of a recursive definition that will be significant later. We define a **string** of words over some vocabulary W

1. Base case:
   If w is a word in W (i.e. $w \in W$), then w is a string over W
2. Base case:
   The empty string, ϵ, is a string over W

   (The empty string is just a string containing no words)
3. Recursive step:
   If s is a string and w is a word in W,
   then ws is a string over W
4. Exclusion:
   Nothing else is a string over W, except through this definition

For example, with a vocabulary $W = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, the following can be shown to be strings over W:

- 1 is a string (base case)
- 21 is a string (one recursion)
- 321 is a string (two recursions)
- ...

1.3.6 Inductive Proofs

Inductive proofs are closely related to recursion. They are used to show that some property P holds of elements in some ordered set of objects (e.g. the set of integers).

Inductive proofs always consist of a **base case** showing that P holds of the first element in the ordering; an **inductive hypothesis** stating the assumption that P holds of every element up to the kth one in the ordering; and an **inductive step** proving that if P holds for the first k elements, then it also holds for the first $k+1$ elements.

That is:

- Base case:
  Prove P holds of first element (e.g. 0 if we are looking integers)
- Inductive Hypothesis:
  Assume P holds of the first k elements
- Inductive Step:
  Given that P holds of the first k elements, show it also holds for the $k+1$th element

Taking an inductive proof where P is some property of the integers, the base case shows that that $P(0)$ is true. The inductive step allows us to show from this that $P(1)$ is true. Applying the inductive step again allows us to show that $P(2)$ is true, and so on.

Here is an example of a (numerical inductive proof:

Prove $0 + 1 + \ldots + n = n(n+1)/2$

- Base: $0 = 0(0+1)/2$
- Hyp: Assume $0 + \ldots + n = n(n+1)/2$ for all values of n up to k
- Ind: Prove inductive hypothesis holds for $m = k+1$
  $0 + \ldots + k + (k+1) = [k(k+1)/2] + (k+1)]$ by ind. hyp
  $= (k+1)[(k/2)+1)]$
  $= (k+1)(k+2)/2$
  $= m(m+1)/2$

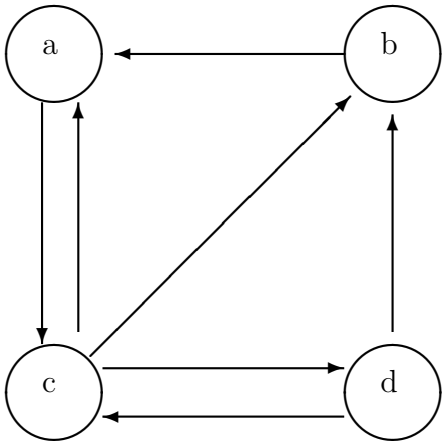
1.3.7 Directed Graphs

Directed graphs are important here (a) for formal analysis of state-transition machines, and (b) for representing trees showing how sentences may be derived from a grammar.

A directed graph is essentially a set of **states** (or **nodes**), N, and a set of **arcs** (or **arrows, transitions**) connecting them. The arcs are represented as a binary relation over N. Thus:

> A directed graph consists of
>
> (a) Set N of nodes
>
> (b) A binary relation, $\delta \subseteq N \times N$
> — the transition relation
>
> There is an arc going from node n_1 to n_2 iff $\delta(n_1, n_2)$

For example:



$$\begin{aligned} N = \quad & \{a, b, c, d\} \\ \delta = \quad & \{\langle a, c\rangle, \langle b, a\rangle, \\ & \langle c, a\rangle, \langle c, b\rangle, \langle c, d\rangle, \\ & \langle d, b\rangle, \langle d, c\rangle\} \end{aligned}$$

Some more terminology to do with graphs:

- A **path** is a sequence of nodes where each is connected to the next by a transition
- A **null path** is a path of length zero connecting a node to itself
- A **cycle** is a path of length one or more that connects a node to itself
- Directed graphs with no cyclic paths are called **Directed Acyclic Graphs** (DAGs).

Trees

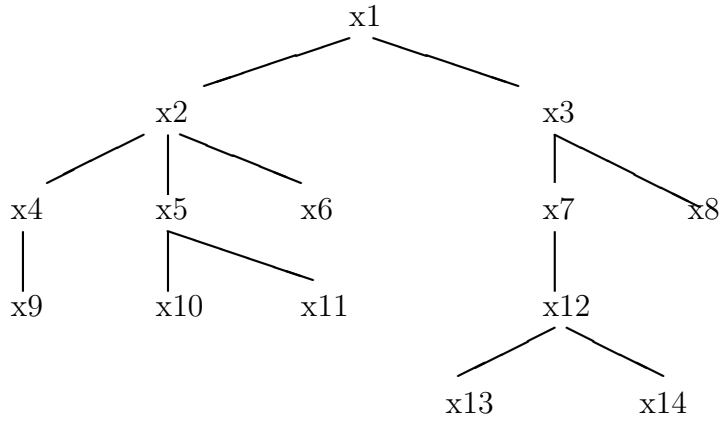
A tree is a directed acyclic graph, where

(a) Each node (except for the root node) has exactly one transition coming into it.

(b) The **root** node of the tree has no transitions coming into it

(c) For each node, there is a unique path going from the root to the node.

An example of a tree is



Note that the arrow head on the transitions are omitted — they would point downwards. The root node of this tree is $x1$

- **Leaf** nodes are the ones with no transitions (branches) coming out of them
  i.e. $x9, x10, x11, x6, x13, x14, x8$
- $x4, x5$ and $x6$ are **daughters** of their mother $x2$,
  and are sisters of each other
- $x2$ is the root of a subtree of $x1$, where the subtree contains $x2$ and all the nodes below it — the **descendants** of $x2$
- Likewise, $x5$ is the root of a subtree of $x2$

1.4 Summary: strings, languages, grammars and machines

A vocabulary W is a finite set of words. A string over W is an ordered sequence of words taken from W. It can include the empty string ϵ, and may repeat words.

The set of all strings over a vocabulary W is written as W^*, and is an infinite set. W^* may be defined recursively as

- $\epsilon \in W^*$
- If $w \in W$, then $w \in W^*$
- If $s \in W^*$ and $w \in W$, then $ws \in W^*$
- Nothing else is a string over W, except through this definition

A language L over vocabulary W is a set of strings, such that $L \subseteq W^*$. The members of L are known as sentences of the language.

A formal language is one for which there is a finitely specifiable set of rules for deciding which strings are and are not sentences of the language. The rules are usually given by means of either a grammar or an automaton.

A grammar is a finite set of rules operating on two disjoint sets of symbols: the **variables** (or non-terminal symbols), V, and the **words** (or terminal symbols) W. Rules take the form

$$LHS \rightarrow RHS$$

where $LHS \in (W \cup V)^*$ and $RHS \in (W \cup V)^*$

Different classes of grammar emerge from imposing varying restrictions on the strings of symbols that occur on the left and right hand sides of rules. A grammar always has a distinguished variables symbol, known as the start symbol.

> **Notational Convention**
> When writing abstract grammars down, there is a general convention that variable symbols begin with or are an upper case letter, words begin with or are a lower case letter, and that the start symbol is S.

Sentences are derived from a grammar by writing down a string consisting solely of the start symbol. Grammar rules are repeatedly applied to rewrite portions of the string until one finally obtains a string containing just words.

An automaton is a state-transition machine. It must have a finite number of states and transitions, with a single deisgnated start state and one or more designated accepting states. Strings are accepted by the automaton if they trigger a series of transitions from the start state and ending in an accepting state. Different classes of automaton have different degrees of memory associated with them.

Part I

Regular Languages

In the following three chapters we will begin at the bottom of the Chomsky hierarchy, looking at regular languages. Regular languages can be defined either by means of regular grammars and expressions, or by finite state automata. Chapter 2 introduces finite state automata, and shows how they can be used to define languages. Chapter 3 introduces regular grammars and expressions, and shows how they too can be used to define languages. More specifically, we will see how finite state automata and regular grammars can be used to specify the same languages. The final chapter in this part demonstrates that not all languages are regular, and hence cannot all be defined either by finite state automata or regular grammars.

Regular languages have one main advantage, and one main disadvantage. The advantage is that the question of whether a particular string belongs to a given regular language can be computed very efficiently indeed. This makes regular languages very useful in applications such as pattern matching. The `grep` utility in Unix, for example, searches for strings belonging to a user-defined regular language (specified by means of a regular expression).

The disadvantage of regular languages is that they are not very expressive. They are not expressive enough to serve as programming languages, for example; context free languages are used for this.

Chapter 2

Finite State Automata

This chapter is organised as follows. Section 2.1 introduces the simplest type of finite state automaton, known as a **deterministic finite state automaton** (DFA). Section 2.2 introduces a more complex type of automaton, known as a **non-deterministic finite state automaton** (NFA). Section 2.5 shows that NFAs and DFAs are equivalent: from any NFA you can derive a DFA that accepts exactly the same language.

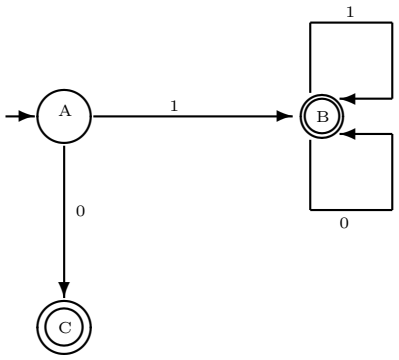
2.1 Deterministic Finite State Automata

A finite state automaton consists of a finite set of **states**, with arrows or **transitions** connecting the states. The transitions are labelled with words from a given vocabulary. The automaton must have a single designated **start state**, and a designated set of one or more **accepting states**.

This characterisation is true both of the deterministic finite state automata (DFAs) described in this section, and of the non-determininstic finite state automata (DFAs) described in section 2.2

2.1.1 Examples of (Deterministic) Finite State Automata

Repeated below is a detreministic finite state automaton for accepting binary numbers without leading zeroes.



It comprises three states, A, B and C, of which A is the designated start state, and B and C are the accepting states. In diagrams like this, the start state is always indicated by having a short unlabelled incoming transition. The accepting states are represented as double circles.

The transitions between states can usefully represented as a transition table:

	0	1
A	C	B
B	B	B
C	–	–

The first line says that there is a 0-transition from state A to state C, and a 1-transition from state A to state B. The second line says that 0- and 1-transitions loop from state B back onto state B. The third line says that there are no transitions out of state C.

Accepting Sentences of the Language

To use the automaton to show that the string 1010 is a sentence of the language, begin in the start state. Look at the first word in the string, which is 1. Take the transition labelled with a 1 to the next state (i.e. state B). Look at the next word in the string, which is 0. Take the transition from state B labelled with a 0. This loops back to state B. Look at the next word (1) and take the 1-transition from B to B. Look at the next (and final) word, 0. Take the 0-transition from B to B. We have now got to the end of the string, and we are in an accepting state. This means that the string is a sentence in the language defined by the automaton.

To show that 011 is not a string in the language, again begin in the start state A. The first word is 0, so we have to take the 0-transition to state C. The next word is 1, but there are no 1-transitions leaving state C. In other words, we cannot read through to the end of the string to arrive in an accepting state, and so the string is not a sentence of the language.

This procedure for accepting (or rejecting) strings as being sentences of the language defined by the machine can be presented algorithmically as follows.

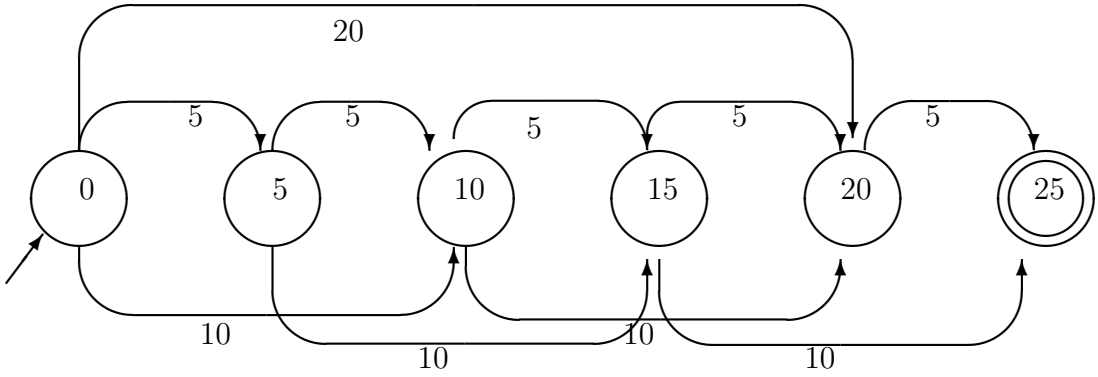
DFA Acceptance:

1. Set current state, *curr*, to start state,
   and input *inp* to string to be accepted
2. If *curr* is an accepting state
   and input *inp* is empty
   ACCEPT (and exit)
3. If input is empty and *curr* is not an accepting state
   REJECT (and exit)
4. Let *inp* be $word + inp'$, where *word* is the first word of the input, and inp' the remainder.
5. If there is no transition from *curr* using *word*
   REJECT (and exit)
6. Else
   Set *curr* = the state reached by taking the *word* transition from *curr*
   Set $inp = inp'$
   Repeat from step 2

The algorithm for *generating* a sentence is similar to the above, except that we pick an arbitrary path from start to finish, taking the sequence of labels on the transitions as the sentence generated.

Another DFA

Here is another finite state automaton, this time to recognise valid sequences of coin insertions totalling 25p for a vending machine (early vending machines really did implement mechanical finite state automata like this).



The 'words' we use to communicate with the vending machine are 5p, 10p and 20p coins. 'Sentences' in the vending machine language are any sequence of coins whose value totals 25p. To make the diagram clearer, the states of the machine have been labelled with the total amount of money that needs to have been inserted to reach that state.

Determinism

Both the machines we have described here are *deterministic*. That is, if you are sitting in a given state looking at a given word, there will be at most one transition out of the state for that word. That is, the transitions that can be taken are fully determined; there is never a choice of two equally legitimate transitions at any point. The significance of this determinism will become clearer when we consider non-deterministic FSAs in section 2.2.

2.1.2 Formal Definitions

Rather than draw diagrams of finite state automata, it is more convenient to define them mathematically. A (deterministic) finite state machine, DFA, is a 5-tuple

Definition of Deterministic FSA, DFA

$$DFA = \langle N, V, \delta, s, F \rangle$$

where

- N is a set of states (or nodes)
- V is a set of labels — the input vocabulary of the DFA
- δ is a (labelled) transition function, $\delta : N \times V \mapsto N$
- $s \in N$ is the start state
- $F \subseteq N$ is a set of final states

That is, we have a set of states N, which contains a single starting state s and one or more accepting states contained in F. The transitions between states are represented by the transition function δ. Given a state n and a word w, the transition function will return a new state $n' = \delta(n, w)$. This is the state you can move to from n when the next word is w.

The transition function δ is partial. This means that there can be some state/word combinations that have no transitions defined. If you find yourself in a state looking at a word for which no transition is defined, you cannot move any further through the automaton: you are stuck.

Formal Example

If we were to give a formal definition of the vending machine FSA, it would be

$$Vend = \langle N, V, \delta, s, F \rangle$$

where

- $N = \{s0, s5, s10, s15, s20, s25\}$
- $V = \{5, 10, 20\}$
- $s = s0$
- $F = \{s25\}$
- $\delta = \{\langle s0, 5, s5\rangle, \langle s0, 10, s10\rangle, \ldots, \langle s5, 5, s10\rangle \ldots \langle s20, 5, s25\rangle\}$

We could also represent the transition function as a transition table

δ	5	10	20
s0	s5	s10	s20
s5	s10	s15	s25
s10	s15	s20	—
s15	s20	s25	—
s20	s25	—	—
s25	—	—	—

Extended Transition Function

Having stated how a deterministic FSA is formally defined, we now want to move towards a formal definition of the language accepted by the FSA. Recall that sentences of the language will be any sequence of words that allows you to make transitions from the start state to an accepting state, and where the transitions consume all the words in the process.

As a first step towards a formal definition of this, we need to define an *extended transition function*, written as δ^*. The extended transition function allows you to take several transitions in one go, consuming a string of words as you do so. That is, it is a function that takes a state and a string of words, and returns a state:

$$\delta^* : N \times V^* \mapsto N$$

(Recall that V^* denotes the set of all strings that can be constructed from the vocabulary V)

For example, we might have:

$$\delta^*(n_0, w_1 w_2 \ldots w_i) = n_i$$

where n_0 is some initial state, and $w_1 w_2 \ldots w_i$ a sequence of words. The resulting state n_i is the one reached by taking one transition after another, consuming a word at each go. That is

$\delta(n_0, w_1) = n_1$
$\delta(n_1, w_2) = n_2$
$\ldots \delta(n_{i-1}, w_i) = n_i$

In other words, extended transitions just compose individual transitions together.

The extended transition function may be derived recursively from the basic transition function as follows:

Definition of extended transition function, δ^*

1. Base case
   If $\delta(n_1, w) = n_2)$
   then $\delta^*(n_1, w) = n_2)$
2. Recursion
   If $\delta^*(n_1, v) = n_i$ and $\delta(n_i, w) = n_j$
   then $\delta^*(n_1, vw) = n_j$

Here, we use v to indicate a string of words (which can include strings of just one word). We use w to indicate single words.

Language Accepted by a DFA

Having defined the extended transition function for DFAs, it is straightforward to define the language accepted by a DFA. The language is going to be the set of strings (sentences) accepted by the machine. A string v is accepted by the machine if

$$\delta^*(s, v) \subseteq F$$

That is, you can take an extended transition from the start state s, consuming the whole of the string v, and land up in one of the final accepting states contained in F.

More formally

Language accepted by machine DFA M, $L(M)$

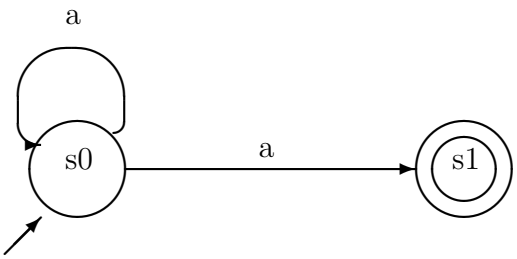
$$L(M) = \{v \mid \delta^*(s, v) \in F\}$$

where F is the set of final accepting states, and s the start state.

Note how we write $L(M)$ for the language defined by a machine M.

2.2 Non-Deterministic FSAs

We now turn to non-deterministic FSAs, or NFAs. The following is the diagram for an NFA that accepts strings consisting of one or more a's:



Note how there are two transitions labelled a coming out of state $s0$. This means that if you are sitting in state $s0$, with a as the next word in the input, you have a choice of two possible transitions to take. This is what makes the automaton non-deterministic.

Consider what happens when you run this machine on the string aaa. Starting in state $s0$ and looking at the first a, we have a choice of either taking the transition that loops back onto $s0$, or the one that goes to $s1$. Suppose we take the transition to $s1$. We are now sitting in $s1$ with aa remaining in the input. But there are no a-transitions out of $s1$, so we are stuck.

But suppose instead we had looped back onto $s0$ with the transition on the first a. Then there would still be an a-transition to consume the second a. Suppose we take the transition that loops back to state $s0$. Then we have a choice of two transitions out of $s0$ for consuming the final a in the input. Suppose this time we take the a-transition to state $s1$. After this, we find ourselves in an accepting state with no remaining input: the string has been accepted.

To summarise, for a string to be accepted by an NFA, there must be at least one sequence of transitions from the start state that leads to a final state and no remaining input. A string is rejected only if there is no such sequence. The fact that there may be other sequences of transitions that do not lead to a terminating condition does not mean that the string is rejected.

2.2.1 Non-Determinism and Backtracking

The algorithm for deciding whether an NFA accepts a string involves *backtracking*. At any point where more than one transition is possible, the algorithm has to make a blind choice about which transition to take. Some choices may lead to successful termination and acceptance, others not. If the algorithm makes the wrong choice, it needs to be able to retrace its steps to the point where the wrong choice was made, and instead try an alternative transition. This retracing of steps and then trying again is known as backtracking.

To see how backtracking works, we need to define some subsidiary notions. A *machine configuration* is a pair consisting of (a) the state the machine is currently in, and (b) the input remaining.

To implement backtracking, the algorithm needs to maintain a stack[1] of choice points. Whenever more than one transition is possible, one of them is chosen,

[1]Stacks are a commonly occurring data structure in computer science. The stock example of a stack is the spring loaded stack of plates you get in some cafeterias. Plates can only be

and the remaining alternatives are pushed onto the stack.

Whenever the machine gets stuck, instead of rejecting the string outright, the algorithm does the following. It pops the previous choice point off the stack, and starts to explore alternative transitions. A string is only rejected when

(a) the stack is empty and

(b) the machine is unable to make a transition out of a non-accepting state, or unable to make a transition from an accepting state with input still to process

The following is an algorithm[2] for accepting strings using an NFA. Note that it is somewhat more complex than the equivalent algorithm for deterministic automata.

Acceptance with an NFA

1. INIT: set current state $curr$ to the start state s
   set input inp to string to recognise
   set stack to empty

2. ACCEPT if in a final state with no remaining input,

3. else REJECT: if stack is empty, there are no possible transitions, and we are not in a final state with no remaining input

4. else TRANSITION:
   if inp is $word + inp'$
   and $S = \{s \mid \delta(curr, word, s)\} \neq \{\}$
   then
   (a) select a state s from S, and set $S' = S - \{s\}$
   (b) PUSH configuration $\langle curr, S', inp' \rangle$ onto stack
   (c) set $curr = s$ and $inp = inp'$
   (d) repeat from step 2

5. else BACKTRACK: if non-empty stack, but no possible transitions and not in final state with no input, then
   (a) POP configuration $\langle old, oldS, oldinp \rangle$ from stack
   (b) select state s from $oldS$, and set $oldS' = oldS - \{s\}$

pushed onto the top of the stack, and can only be popped off the top of the stack. You cannot remove plates from the middle of the stack. Popping and pushing only on the top of the stack means that it is a Last-In-First-Out (LIFO) data structure: You always pop off the last item that was pushed in.

[2]Information only

(c) if $oldS' \neq \{\}$
PUSH $\langle old, oldS', oldinp\rangle$ onto stack

(d) set $curr = s'$ and $inp = oldinp$

(e) repeat from step 2

(Note that we are treating δ as a transition *relation* rather than a transition function. Hence $\delta(curr, word, s)$ expresses that fact that you can make a transition from *curr* to s via *word*. But there may be other states, s', for which the same holds: $\delta(curr, word, s')$. See below.)

2.2.2 Formal Definitions

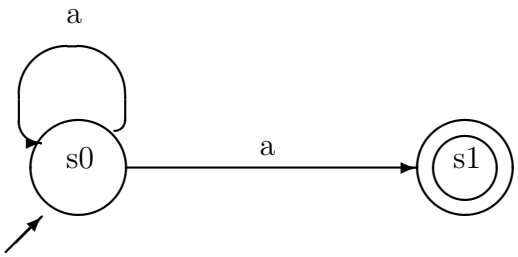
Definition of Non-deterministic FSA, NFA

$$NFA = \langle N, V, \delta, s, F\rangle$$

where

- N is a set of states (or nodes)
- V is a set of labels — the input vocabulary of the NFA
- δ is a (labelled) transition relation, $\delta \subseteq N \times V \times N$
- $s \in N$ is the start state
- $F \subseteq N$ is a set of final states

This is identical to the formal definition of a deterministic FSA, except that δ is a transition relation rather than a transition function. Consider once again the NFA accepting strings of one or more a's



The transition relation for this machine is

$$\delta = \{\langle s0, a, s0\rangle, \langle s0, a, s1\rangle\}$$

Given a relation like this, we can always recast it as a function onto sets of state. In tabular form:

δ	a
$s0$	$\{s0, s1\}$
$s1$	$\{\}$

Or put another way:

$$\delta(s0, a) = \{s0, s1\}$$
$$\delta(10, a) = \{\}$$

Extended Transitions, δ^*

Just as for DFAs, we can construct an extended version of δ. Taking the non-deterministic version of δ to be a function from states and words to sets of states, δ^* will be a function from states and strings of words to sets of states:

$$\delta^* : N \times V^* \mapsto \mathcal{P}(N)$$

That is, if

$$\delta^*(s_1, v) = \{s_i, s_j, \ldots, s_n\}$$

then by taking a sequence of transitions starting at s_1 and consuming the words in v, you can end up in any one of the states $\{s_i, s_j, \ldots, s_n\}$

The extended (non-deterministic) transition function δ^* can be recursively defined in terms of δ:

Definition of extended non-deterministic transition function, δ^*

1. Base case
   $\delta^*(n, w) = \delta(n, w)$
2. Recursion
   If $\delta^*(n, v) = S$
   then $\delta^*(n, vw) = \bigcup\{S' \mid \exists n' \in S \text{ and } \delta(n', w) = S'\}$

Here again, we use v to indicate a string of words (which can include strings of just one word). We use w to indicate single words.

With this definition in place we can give the define the language accepted by a NFA M, $L(M)$ in analagous way to deterministic automata:

Language accepted by NFA M, $L(M)$

$$L(M) = \{v \mid \delta^*(s, v) \cap F \neq \{\}\}$$

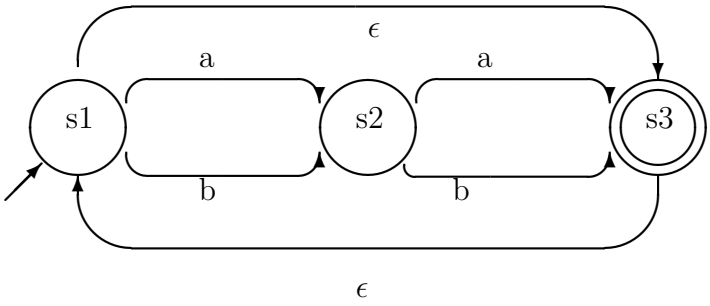
where F is the set of final states, and s the start state.

Note that here we are interested in strings v where there is at least one sequence of transitions from the start to a finish state.

2.3 Machines with Empty (ϵ) Transitions

So far we gave assumed that all transitions are labelled with words from the input vocabulary. However, another kind of transition is possible. This is an empty-, or ϵ-transition. You can take an empty transition from one state to another without consuming a word from the input.

Here, for example is a machine to accept even length strings of a's and b's:



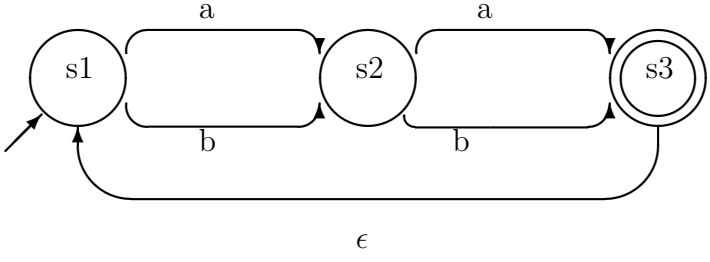
There is an empty transition from state s to s3. This means that you can get from the start state to a final accepting state without consuming any input. This means that the machine will accept the zero length empyt string, ϵ. There is also an empty transition from s3 back to s. This means that having consumed an even length sequence of a's and b's to get to state s3, you can go back to state s, and add two more words to the string.

The presence of the empty transition from state s in the machine above is enough to make it non-deterministic. To see why, suppose we are sitting in state s, with b as the next word on the input. There are two transitions one can take. Either the b-transition to state s2, removing the b from the input. Or the empty transition to state s3, leaving the b on the input.

So, for example, to accept the string *ab*, we could make an *a*-transition to s2 followed by a *b*-transition to s3. Or we could make an empty transition to s3, another empty transistion back to s, then take an *a*- and a *b*-transition.

More generally, if a machine contains any state that has more than one transition coming out of it, and where one of the transitions is empty, then the machine is non-deterministic.

But the bare presence of empty transitions is not on its own sufficient to make a machine non-deterministic. Consider the following:



This has an empty transition out of s3. But this is the only transition out of s3, and the machine is deterministic. The machine accepts even length sequences of a's and b's, but does not accept the empty string.

At a formal level, to permit empty transitions, we do the following. We add ϵ as an additional symbol that can label transitions. The transition functions δ (for non-deterministic machines) becomes a function from states and words plus ϵ to sets of states:

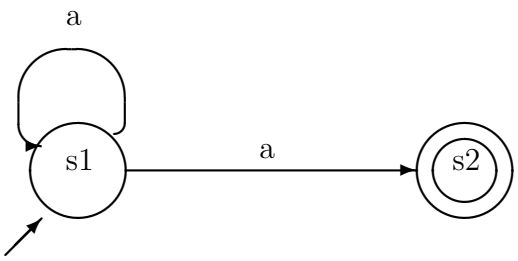
$$\delta : N \times V \cup \{\epsilon\} \mapsto \mathcal{P}(N)$$

2.4 Machines with Error States

An error condition arises in an FSA if we find ourselves sitting in a state s, looking at the next word of the input, a, and with no a-transitions coming out of s. If the machine is deterministic, this means that whatever string brought about this condition is rejected. If the machine is non-deterministic, we have to backtrack tom some earlier configuration to see if the string can be accepted.

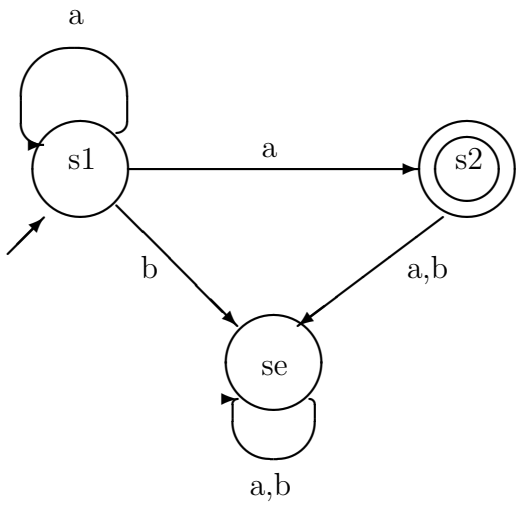
There is an alternative way of dealing with error conditions. The machine provides an a-transition to a single error state. This error state, s_e, has transitions for each word in the vocabulary that loops from s_e back to s_e. But it contains no transitions back to any other state in the machine. Once we reach an error state, the machine loops round consuming the rest of the input, but never leaving the error state. The error condition now becomes that we find ourselves sitting in the error state with no remaining input.

For example, consider the following machine:



What happens if the input string is ba? (Note: assume that the input vocabulary is $\{a, b\}$, but we wish to reject any strings containing a b.) We get stuck in state s1 and an error condition arises.

The following is equivalent to the above in terms of the strings it accepts and rejects, but contains an error state:



With input ba, we make a b-transition to the error state se, and consume the remaining a by looping back to the error state.

It is easy to convert any FSA without an error state into one that has an error state. For every state / word pair, $\langle s, w \rangle$ for which no transition is defined, add a w-transition to the error state. Add looping transitions on the error state for every word in the input vocabulary.

2.5 Eliminating Non-Determinism

In the preceding sections we have looked at a variety of different types of finite state automata

- Deterministic FSAs
- Non-deterministic FSAs
- FSAs with ϵ-transitions
- FSAs with error states

Perhaps surprisingly, all these different types of FSA turn out to be equivalent, in the following sense. For any machine M of one type accepting a language $L(M)$, one can construct a machine M' of another types that accepts exactly the same language.

In the case of machines with and without error states, it is relatively easy to see that this is so. We have slready shown how to add an error state to a machine that lacks one. And removing an error state from a machine is a simple inverse of this.

But in the case of deterministic and non-deterministic FSAs, the equivalence is a little harder to establish, and also rather more surprising. At first sight, one would have thought that non-deterministic FSAs are more complex. They

certainly have a more complicated algorithm defining acceptance of a string. Therefore, one might have thought that NFAs are more expressive in terms of the class of languages they can define. But this turns out not to be so.

The equivalence between DFAs and NFAs is established by defining an algorithm that can convert any NFA into a DFA accepting the same language. (Converting a DFA to an equivalent NFA is trivial: just add some ϵ-transitions that loop back onto the state they started from).

We will first present the conversion algorithm. Having done this, we will prove that the algorithm really does ensure that the two machines accept the same language.

2.5.1 NFA→DFA Conversion Algorithm

The algorithm for converting a non-deterministic FSA to a deterministic one proceeds by creating DFA states out of sets of NFA states. In outline it proceeds as follows:

1 Start state in DFA = set containing (i) start state of NFA plus (ii) all states reachable from it by ϵ-transitions.
Name this state by the set of NFA states it contains

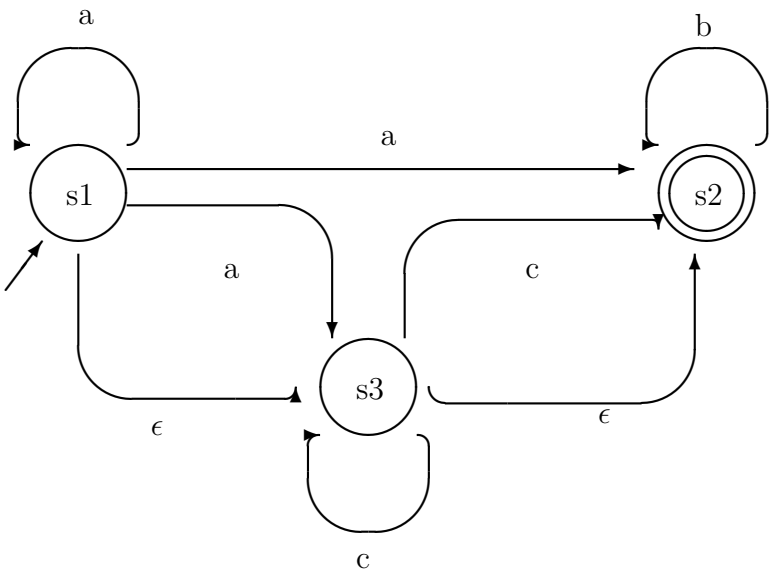
2 For each DFA state, S created
for each input symbol, a,

- collect the set of all states a-reachable from a state in S
- if such a state S' has not already been created, create it, and name it by the set of NFA states it contains
- add a single a-transition from S to S' (if there is not one already there)

3 Repeat (2) until no more new states or transitions can be added.

4 Any DFA state containing an NFA final state is to be counted as a DFA final state.

The operation of this algorithm can be illustrated with the following NFA:



We start by creating the start state for the DFA. We form a set consisting of the DFA start state, $s1$, and all states reachable from it by following only ϵ-transitions. It turns out that both states $s2$ and $s3$ can be reached from $s1$ by taking only ϵ-transitions. Therefore, the start state of the DFA is (labelled($\{s1, s2, s3\}$.

Having identified the start state, we need to identify which NFA states are a-, b-, and c-reachable out of any of the NFA states in $\{s1, s2, s3\}$. Note that one state, s_j is a-reachable from a state s_i if you can take any number of empty transitions from s_i, leading to an intermediate state s_k, followed by an a-transition from s_k to s_{k+1}, followed by any number of empty transitions to reach s_j.

Consider first the NFA states that are a-reachable from any state in $\{s1, s2, s3\}$. From state $s1$, we can take a single a-transition to get any one of $s1$, $s2$ or $s3$. We can also take an $a$transition to $s3$ followed by an empty transition to s_2. But from s_2 and s_3, no states are a-reachable. Therefore, the set of all states a-reachable from any state in $\{s1, s2, s3\}$ is in fact $\{s1, s2, s3\}$.

Now consider the states that are b-reachable from any state in $\{s1, s2, s3\}$. From state $s1$ we can take two empty transitions, from $s1$ to $s3$ and then from $s3$ to $s2$, and then take a b-transition looping back to $s2$. From state $s2$, we can take a single b-transition looping back to $s2$. And from $s3$ we can take an empty transition to $s2$, followed by a b-transition looping back to $s2$. No other states are b-reachable, so the set of NFA states b-reachable from any NFA state in $\{s1, s2, s3\}$ is $\{s2\}$.

Now consider the c-reachable states from $\{s1, s2, s3\}$. From $s1$ we can take an empty transition to $s3$, followed by a c-transition looping back to $s3$. We can also take an empty transition to $s3$ and then a c-transition to $s2$. No state is c-reachable from $s2$. From state $s3$ we can take a c-transition looping back to $s3$, or a c-transition to $s2$, or a c-transition looping back to $s3$ followed by an empty transition to $s2$. Hence the set of NFA states c-reachable from any NFA state in $\{s1, s2, s3\}$ is $\{s2, s3\}$.

We can enter this in a transition table as follows

DFA State	Transition	DFA State
$\{s1, s2, s3\}$	$\xrightarrow{a}$	$\{s1, s2, s3\}$
	$\xrightarrow{b}$	$\{s2\}$
	$\xrightarrow{c}$	$\{s2, s3\}$

Looking at this table, we can see that we have constructed two new DFA states, $\{s2\}$ and $\{s2, s3\}$. We therefore need to determine the sets of a-, b- and c-reachable states from these two new states.

Taking $\{s2\}$ first, no states are a- or c-reachable from $s2$. And only $s2$ is b-reachable from $s2$. Thus we add another line to the transition table:

DFA State	Transition	DFA State
$\{s1, s2, s3\}$	$\xrightarrow{a}$	$\{s1, s2, s3\}$
	$\xrightarrow{b}$	$\{s2\}$
	$\xrightarrow{c}$	$\{s2, s3\}$
$\{s2\}$	$\xrightarrow{a}$	$\{\}$
	$\xrightarrow{b}$	$\{s2\}$
	$\xrightarrow{c}$	$\{\}$

This constructs one new state, $\{\}$ (which will be an error state).

Now taking the state $\{s2, s3\}$, we can determine the third line of the transition table:

DFA State	Transition	DFA State
$\{s1, s2, s3\}$	$\xrightarrow{a}$	$\{s1, s2, s3\}$
	$\xrightarrow{b}$	$\{s2\}$
	$\xrightarrow{c}$	$\{s2, s3\}$
$\{s2\}$	$\xrightarrow{a}$	$\{\}$
	$\xrightarrow{b}$	$\{s2\}$
	$\xrightarrow{c}$	$\{\}$
$\{s2, s3\}$	$\xrightarrow{a}$	$\{\}$
	$\xrightarrow{b}$	$\{s2\}$
	$\xrightarrow{c}$	$\{s2, s3\}$

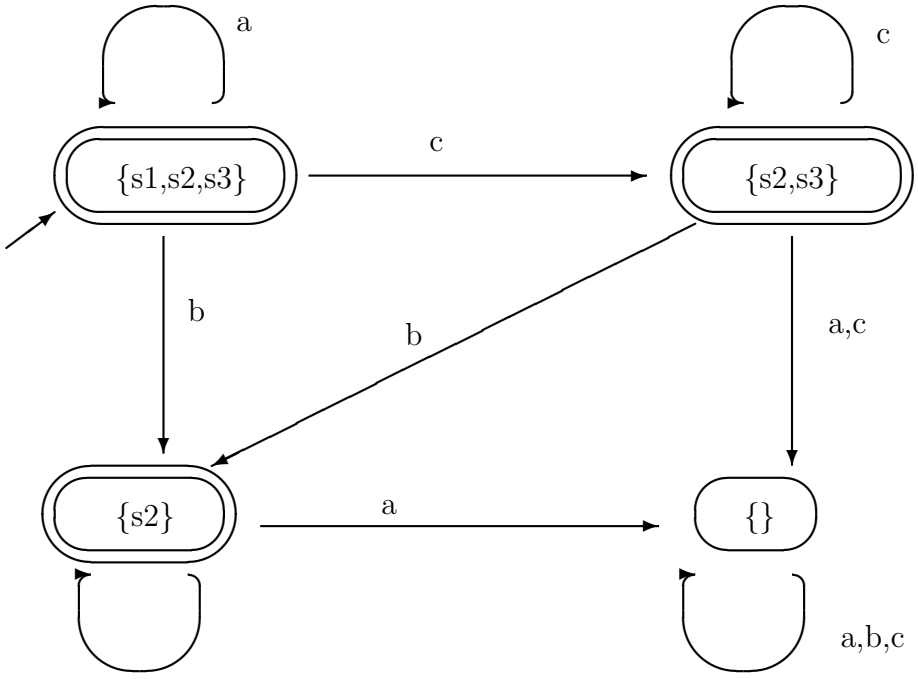
This adds no further new states.

Finally, we have to consider transitions out of the state $\{\}$. Since this contains no NFA states, no NFA states are reachable from any state within it, and so the last line of the transition table can be added:

DFA State	Transition	DFA State
$\{s1, s2, s3\}$	$\xrightarrow{a}$	$\{s1, s2, s3\}$
	$\xrightarrow{b}$	$\{s2\}$
	$\xrightarrow{c}$	$\{s2, s3\}$
$\{s2\}$	$\xrightarrow{a}$	$\{\}$
	$\xrightarrow{b}$	$\{s2\}$
	$\xrightarrow{c}$	$\{\}$
$\{s2, s3\}$	$\xrightarrow{a}$	$\{\}$
	$\xrightarrow{b}$	$\{s2\}$
	$\xrightarrow{c}$	$\{s2, s3\}$
$\{\}$	$\xrightarrow{a}$	$\{\}$
	$\xrightarrow{b}$	$\{\}$
	$\xrightarrow{c}$	$\{\}$

This now takes care of all transitions out of the DFA states we have created. It remains only to identify the final accepting states in the DFA. This will be any state containing an NFA final state. i.e $s2$.

We can therefore draw the DFA derived from our original NFA as follows



2.5.2 Formal Proof of NFA / DFA Equivalence

The last section illustrated how to determinise an NFA. However, we have not established that this procedure actually produces a DFA accepting the same

language as the NFA. The proof that follows is not something you will be expected to regurgitate in an exam, but you should convince yourself that you are able to follow it.

Conversion Process

We start by giving a more formal description of the procedure described above for determinising an NFA.

We start with a non-deterministic FSM

$$N = \langle S, V, \delta, s, F \rangle$$

and we want to create a deterministic machine

$$D = \langle S', V, \delta', s', F' \rangle$$

1. First build up the states, S', in the new machine D

   We do this recursively, starting with the construction of the new initial state, s'.

   Base Case:

   > s' is the set of states in N that can be reached from s without consuming any input.
   >
   > $s' = \{s_i \mid s_i \in \delta^*(s, \epsilon)\}$
   >
   > Add s' to S', so that
   >
   > $s' \in S'$

   Recursion:

   > Suppose S_k is a state in S'.
   > This means that S_k will consist of a set of states from N:
   >
   > $$S_k = \{s_{k1}, s_{k2}, \ldots, s_{kn}\}$$
   >
   > For each word in the vocabulary, a, collect the set of states S_{la}, where
   >
   > $$S_{la} = \{s_l \in S \mid s_l \in \delta^*(s_{ki}, a) \text{ for some } s_{ki} \in S_k\}$$
   >
   > That is, the set of states that can be reached in N starting from some state in $s_{ki} \in S_k$ and consuming just one word of input.
   > (Note that there may be some empty transitions taken in going from a state s_{ki} to a state s_l).
   > Ensure $S_{la} \in S'$ by adding it to S' if it is not already there.
   > Repeat for the new state S_l added, and carry on until no more new states can be added.

2. Now we build up the transitions.

   Let S_k be a state in S', where $S_k = \{s_{k1}, s_{k2}, \ldots, s_{kn}\}$

   Let S_{la} be the state previously constructed by gathering together all the states that are a-reachable in N from some state $s_{ki} \in S_k$

   $$S_{la} = \{s_l \in S \mid s_l \in \delta^* s_{ki}, a \text{ for some } s_{ki} \in S_k\}$$

   Then add a transition:

   $$\delta'(S_k, a) = S_{la}$$

3. Now pick out the final states

   $S_k \in F'$ iff there is some $s_{ki} \in S_k$ and $s_{ki} \in F$

Proving Equivalence

Having constructed D from N, we now need to show that it accepts the same language, $L(D) = L(M)$

We first of all need to show

Theorem 1

(a) If processing a string u in N can lead to any one of the states $S_{u1}, s_{u2}, \ldots, s_{un}$

(b) Then processing u in D will lead to the single state $\{S_{u1}, s_{u2}, \ldots, s_{un}\}$

(c) And vice versa

Having done this, we can then reason as follows.

1. Show that any string accepted by N is accepted by D

   Suppose that a string v is accepted by N. Processing it will lead to any one of the states $s_{v1}, \ldots, s_{vn}$, at least one of which will be a final state in N.

   Processing v in D will lead to the state $S_v = \{s_{v1}, \ldots, s_{vn}\}$. Since this contains at least one final state from N, S_v is a final state in D.

   Thus v will also be accepted by D.

2. Show that any string accepted by D is accepted by N.

   Processing v in D will lead to the state $S_v = \{s_{v1}, \ldots, s_{vn}\}$. Since this is a final state, it will contain at least one final state from N.

   Processing v in N can lead to any one of $s_{v1}, \ldots, s_{vn}$. Since at least one of these is a final state in N, v is accepted by N.

To prove theorem 1, we use an inductive proof.

Base: Suppose u is the empty string, ϵ.

Then the states that can be reached by processing u in N is

$$\{s_i \mid s_i \in \delta^* s, \epsilon\}$$

By the construction of D, the initial state is also

$$s' = \{s_i \mid s_i \in \delta^* s, \epsilon\}$$

Since D is deterministic (contains no ϵ-transitions), s' is the only state that can be reached in D by processing $u = \epsilon$.

Induction: Let $u = va$, where v is a string of length n.

By the inductive hypothesis, if processing v in N can lead to any one of $s_{v1}, \ldots, s_{vn}$, then processing v in D leads to the state $S_v = \{s_{v1}, \ldots, s_{vn}\}$.

Processing va in N leads to the set of states that are a-reachable from any of the s_{vi}.

Let $S_{va} = \{s \mid s \in \delta^*(s_{vi}, a)$ for some $s_{vi}\}$

By the construction of D, S_{va} is precisely the state that is reached by processing an a from S_v.

Hence the hypothesis holds for $u = va$ if it holds for v.

Proof of converse is similar

2.5.3 Comments on NFA / DFA Equivalence

For the exams, you will need to now how to determinise an NFA using the procedure in section 2.5.1, but will not need to remember the details of the proof of its correctness.

As mentioned, the equivalence is an initially surprising result: non-deterministic FSAs are no more powerful than deterministic FSAs, despite the much simpler acceptance algorithm for the deterministic case. However, all we have really done is shifted the complexity about. This is because in general determinising an NFA will produce a more complex DFA, with more states, and more transitions.

In the worst case, starting with a set of states, S, and a vocabulary V, we can create

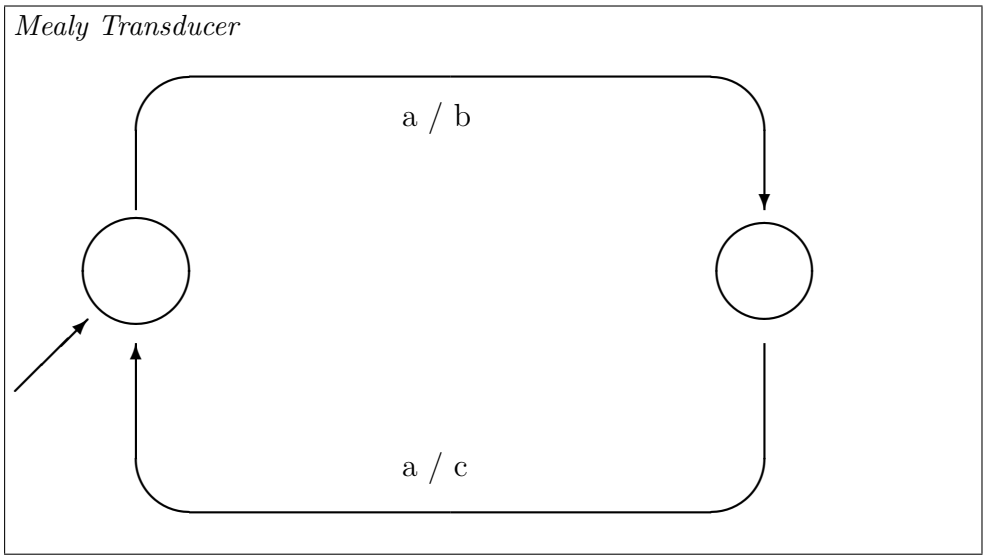
1. $card(\mathcal{P}(S))$ states, and
2. $card(\mathcal{P}(S))card(V)$ transitions

If S and V are finite, then we can however be sure that we will only create a finite number of new states and transitions. Hence, algorithm for converting NFSMs to DFSMs is guaranteed to terminate.

2.6 Finite State Transducers

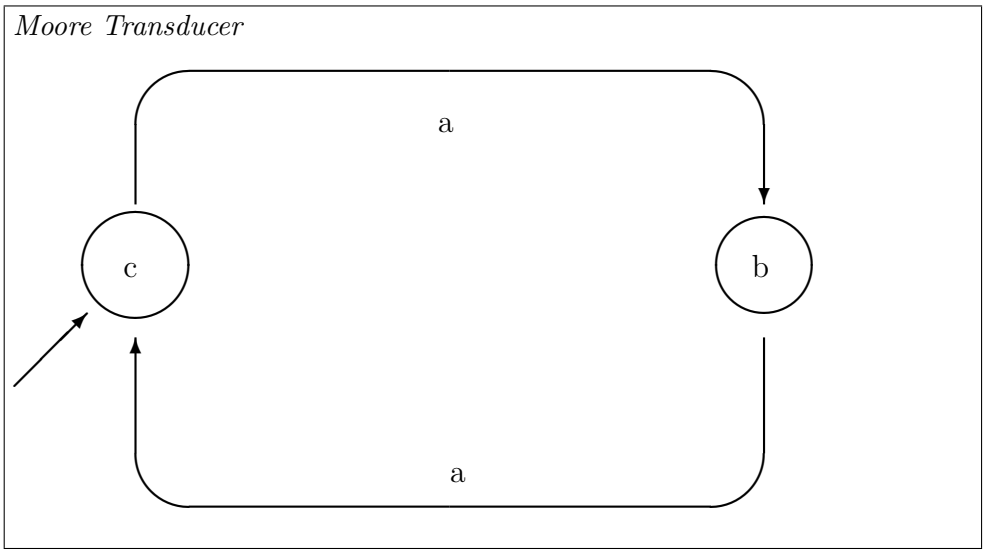
So far we have considered FSAs that accept (or generate) languages. Here we consider *finite state transducers*, which can convert strings of one language to strings of another. Finite state transducers are finite state machines where the by-product of accepting a string in one language is to produce a string in another language.

Here is a finite state transducer that will accept strings consisting of any number of as, and produce an output string that replaces every second a with a c, and every first a with a b:



The transitions are labelled a/b and a/c. This means that to take a transition one has to consume an a from the input, and produce a b (or c) as output. Note that although a start state is indicated, no final accepting state need be shown.

The transducer above is known as a *Mealy machine*: output is produced by transitions. An alternative style of transducer is a *Moore machine*: ooutput is produced by states:



In a Moore machine, an output symbol is produced whenever you enter a state (rather than whenever you take a transition). The Moore machine above will always produce an extra initial c (on entering the start state), but otherwise will exchange every first a for a b, and every second a for a c.

2.6.1 Moore = Mealy

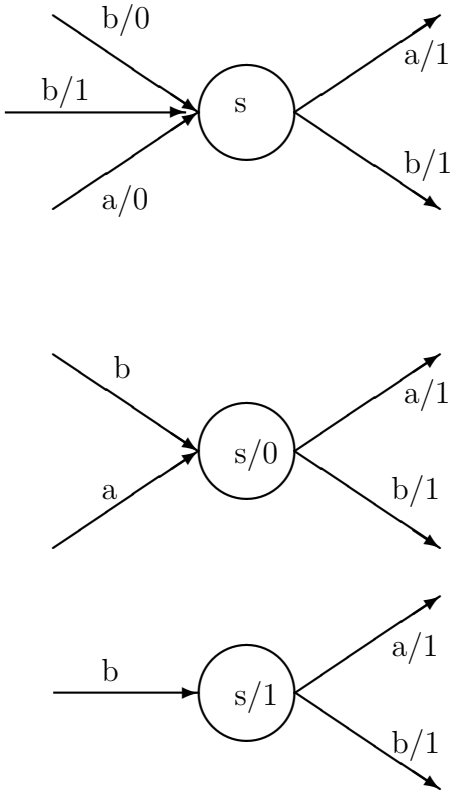
Moore and Mealy transducers are inter-convertible, subject to one slight difference between them: The output on the initial state of a Moore machine means that the length of the output of a Moore machine is always 1 plus the length of the input. But for a Mealy machine, the length of the output is equal to the length of the input. To convert between the two types of transducer, we need to assume that the initial word of output in a Moore machine is a dummy that can be ignored.

Moore to Mealy

To convert a Moore to a Mealy machine, move the output produced by a state to the output on all the arcs coming into the state.

Mealy to Moore

To convert a Mealy to a Moore machine, make a new copy of each state for each diffent output symbol on its incoming transitions.



2.7 Summary

In this chapter, we have introduced several types of finite state automata:

1. Deterministic FSAs
2. Non-deterministic FSAs
3. FSAs with ϵ-transitions
4. FSAs with error states

These all consist of a finite number of states, with a finite number of labelled transitions between them, a designated start state, and a designated set of final accepting states. We have shown how these various machine define languages: the set of strings that will take you from the start state to a final accepting state.

One of the main results is that all the machines above are equivalent: language defined by one type of machine can be defined by a machine of any other type. In particular, non-deterministic FSAs are no more expressive than their deterministic counterparts. This equivalence between deterministic and non-deterministic machines will *not* hold when we come to look at more complex machines, such as push-down stack automata.

The chapter also introduced the idea of finite state transducers, that convert strings in one language to strings in another. Two broadly equivalent classes of finite state transducer were introduced: Moore machines and Mealy machines.

In the next two chapters, we turn to a deeper discussion of the kinds of languages accepted by finite state automata. In chapter 3 we look at alternate ways of defining the same languages, either in terms of regular expressions or regular grammars. And in chapter 4 we look at what kinds of language can and cannot be defined by these means.

Chapter 3

Regular Expressions and Grammars

In the last chapter we introduced various superficially different types of finite state automata, and how these define languages in terms of the set of strings that can take you from a start state to a finish state. We also showed that all the types of finite state automata were equivalent in that one could convert from one type to any other type, while still defining the same language.

In this chapter, we turn to two alternative ways of characterising the languages defined by finite state machines. These are (i) regular expressions, and (ii) regular grammars.

3.1 Regular Sets and Expressions

3.1.1 Regular Sets

Recall that a language is just a set of strings. This means that we can combine languages to form new ones using standard set-theoretic operations: union, intersection, complement, subset, etc.

Regular sets are sets of strings that can be built up using a limited range of set-theoretic operations, starting from a basic stock of languages.

The basic stock of languages comprise

1. The empty set of strings, $\{\}$
2. All singleton sets of one word strings
   $\{a\}, \{b\}, \ldots$, for all words $a, b, \ldots$ in a given vocabulary V
3. The language consisting of just the empty string, $\{\epsilon\}$

   (Note: $\{\}$ and $\{\epsilon\}$ are distinct.
   $\{\epsilon\}$ is the language containing just the empty string
   $\{\}$ is a language containing nothing, not even the empty string)

From these, we can construct new languages using just the following operations

1. Union of languages, $L_1 \cup L_2$
2. Concatentation of languages, $L_1.L_2$
3. Kleene star, L^*

The concatenation of two languages, $L_1.L_2$, is the set of strings that you can obtain by taking one string from L_1, and joining a string from L_2 onto the end of it.

Concatenation of languages, $L_1.L_2$

$$L_1.L_2 = \{s_1 s_2 \mid s_1 \in L_1 \text{ and } s_2 \in L_2\}$$

Examples:

$$\{a\}.\{b\} = \{ab\}$$

$$\{aa, bb\}.\{cc, dd\} = \{aacc, aadd, bbcc, bbdd\}$$

The Kleene star of language is what you obtain by concatenating a language with itself zero, one, two, three etc times, and taking the union of the results

Kleene star of a language, L^*

$$L^* = \{\epsilon\} \cup L \cup L.L \cup (L.L).L \cup ((L.L).L).L \cup \ldots$$

Example

$$\{a\}^* = \{\epsilon, a, aa, aaa, aaaa, \ldots\}$$

More formally, we can give a recursive definition of a regular set over a vocabulary V as follows

Regular sets over a vocabulary V

1. Basis: $\{\}$, $\{\epsilon\}$ and $\{w\}$ (for every $w \in V$) are regular sets over V
2. Recursion: Let X and Y be regular sets over V
   Then the following are also regular sets over V:
   $X \cup Y$ — union/disjunction/or
   $X.Y$ — concatenation/conjunction/and
   X^* — iteration/Kleene star

Here are some examples of regular sets over the vocabulary $\{a, b\}$:

Some regular sets over $\{a, b\}$

Regular Set	Equivalent to
$\{a\}$	$\{a\}$
$\{a\}.\{a\}$	$\{aa\}$
$(\{a\}.\{a\}) \cup \{b\}$	$\{aa, b\}$
$\{a\}.(\{a\} \cup \{b\})$	$\{aa, ab\}$
$(\{a\} \cup \{b\}).\{a\}^*$	$\{a, b, aa, ba, aaa, baa, \ldots\}$

Note that each regular set defines a language over the vocabulary. Thus the regular set $\{a\}.\{a\}$ defines the language $\{aa\}$ Note also that infinite languages can be defined by means of the Kleene star operation.

In due course we will establish that for every regular set over a vocabulary V, there is a corresponding finite state automaton defining the same language, and vice versa.

3.1.2 Regular Expressions

Each regular set defines a language. However, it rapidly gets to be quite tiring writing down all the curly brackets. Regular expressions are an alternative notation for describing regular sets, but without the curly brackets.

To begin with, we write the regular set $\{a\}$ is written $\mathbf{a}$, $\{\}$ as $\mathbf{0}$, and $\{\epsilon\}$ as ϵ. We represent the set operation of union as alternation, | between two regular expressions. Concatenation is represented by just concatenating two regular expressions together. Kleene star is written the same way, but applies to regular expressions rather than sets.

We can thus recast the recursive definition of a regular set to give a definition of regular expressions over some vocabulary V:

Regular expressions over a vocabulary V

1. Basis: $\mathbf{0}, \epsilon$ and $\mathbf{w}$ (for every $w \in V$) are regular expressions over V
2. Recursion: Let X and Y be regular expressions over V
   Then the following are also regular expressions over V:
   $X \mid Y$ — union/disjunction/or
   XY — concatenation/conjunction/and
   X^* — iteration/Kleene star

Taking the previous examples of regular sets over $\{a, b\}$ the corresponding regular expressions are shown below:

Regular Expression	Regular Set
$\mathbf{a}$	$\{a\}$
$\mathbf{ab}$	$\{a\}\{b\} = \{ab\}$
$\mathbf{a}\|\mathbf{b}$	$\{a\} \cup \{b\} = \{a, b\}$
$\mathbf{a}^*$	$\{a\}^* = \{\epsilon, a, aa, \ldots\}$
$(\mathbf{a}\|\mathbf{b})^*$	$(\{\{a\} \cup \{b\}\})^* = \{\epsilon, a, b, aa, ab, ba, bb, \ldots\}$
$(\mathbf{ab})^*$	$(\{a\}.\{b\})^* = \{\epsilon, ab, abab, ababab, \ldots\}$

One further bit of commonly used notation in regular expressions is the + operator, indicating one or more concatentations of a regular expression with itself (cf $*$, which is zero, one or more concatentations). This operation can be defined as follows

$$\mathbf{u}^+ = \mathbf{u}\mathbf{u}^*$$

Regular Expression Identities

Just as every regular set defines a language, so does every regular expressions. However, two different regular expressions can sometimes represent the same language.

There are a number of standard identities between regular expressions (showing when to expressions represent the same language), some of which are shown below:

1. $\mathbf{0u} = \mathbf{u0} = \mathbf{0}$
2. $\epsilon\mathbf{u} = \mathbf{u}\epsilon = \mathbf{u}$
3. $\mathbf{0}^* = \mathbf{0}$
4. $\epsilon^* = \epsilon$
5. $\mathbf{u}^* = (\mathbf{u}^*)^*$
6. $\mathbf{u} \mid \mathbf{0} = \mathbf{u}$
7. $\mathbf{u} \mid \mathbf{u} = \mathbf{u}$
8. $\mathbf{u} \mid \mathbf{v} = \mathbf{v} \mid \mathbf{u}$
9. $\mathbf{u}(\mathbf{v} \mid \mathbf{w}) = \mathbf{uv} \mid \mathbf{uw}$
10. $(\mathbf{u} \mid \mathbf{v})\mathbf{w} = \mathbf{uw} \mid \mathbf{vw}$
11. $(\mathbf{uv})^*\mathbf{u} = \mathbf{u}(\mathbf{vu})^*$
12. $(\mathbf{u} \mid \mathbf{v})^* = (\mathbf{u}^* \mid \mathbf{v})^*$
    $= \mathbf{u}^*(\mathbf{u} \mid \mathbf{v})^* = (\mathbf{u} \mid \mathbf{vu}^*)^*$
    $= (\mathbf{u}^*\mathbf{v}^*)^* = \mathbf{u}^*(\mathbf{vu}^*)^*$
    $= (\mathbf{u}^*\mathbf{vu}^*)^*$

These identities, plus a certain amount of common sense, can be used to simplify regular expressions. For instance

> Simplify the regular expression $\mathbf{a}(\mathbf{aa})^*(\epsilon \mid \mathbf{a})\mathbf{b} \mid \mathbf{b}$
>
> Note that $\mathbf{a}(\mathbf{aa})^*$ corresponds to any odd number of a's
> Then $\mathbf{a}(\mathbf{aa})^*(\epsilon \mid \mathbf{a})$ corresponds to any odd number of a's followed by another a or nothing. That is, any number of a's greater than one
> That is $\mathbf{a}(\mathbf{aa})^*(\epsilon \mid \mathbf{a}) = \mathbf{a}^+$
> Hence $\mathbf{a}(\mathbf{aa})^*(\epsilon \mid \mathbf{a})\mathbf{b} \mid \mathbf{b} = \mathbf{a}^+\mathbf{b} \mid \mathbf{b}$
> But $\mathbf{a}^+\mathbf{b} \mid \mathbf{b} = \mathbf{a}^*\mathbf{b}$

Simplifications like this are a common exam question.

3.2 Regular Expressions in Unix

Regular expressions are widely used in Unix, in particular for pattern mapping operations. For example, the `grep` facility can be used to search for strings matching a specified regular expression. However, the syntax used for regular expressions in Unix differs in some small respects from the mathematical notation used above. Unix regular expressions also allow some constructs that go slightly beyond what is strictly allowed by regular expressions.

3.2.1 `regex`

A full description of Unix regular expressions may be obtained by looking up the relevant `man` page:

```
unix-prompt% man -s 5 regex
```

Here we just review some of the salient points.

Single Character Matches

- Most characters match themelves
- `.` matches any single character
- Exceptions: special characters `*  [[ \ ^ $`
  — need to be preceded by a backslash
- Bracket expressions — enclosed between `[ ]`:
  - Matching list
    `[abc]` — matches `a`, `b` or `c`
  - Non-matching list
    `[^abc]` — matches any chcracter but `a`, `b` or `c`
  - Collating sequence (only works in bracket exprs)
    `[abc[.ch.]]` — matches `a`, `b`, `c` or `ch`
    `ch` is collated into a single character
  - Ranges
    `[0-9]` — matches `0`, `1`, `2`, . . . , `9`

Multiple Character Matches

- Concatenation of regular expressions
- Subexpressions enclosed by `\(...\)`
- Backreference: `\n` backreference to nth subexpression
  e.g. `\(a.c\)d\1` — two occurrences of `a` something `c` with an intervening `d`

- Repetitions, may follow single characters, subexpressions or backreferences
  - `*` zero or more repetitions
  - `\{m\}` exactly m repetitions
  - `\{m,\}` at least m repetitions
  - `\{m,n\}` between m and n repetitions
- Example: `\(abc\)\{2,3\}` — two or three repetitions of `abc`

Anchors

- `^`: anchors string to start of line
- `$`: anchors string to end of line
- Example: `^\(.*\)\1$` — matches any line consisting of identical first and second halves

The facility to match on backreferences takes Unix regular expressions beyond what can be done according to the strict mathematical definition of regular expressions. However, the significance of this comment will only become clear in the next chapter.

3.2.2 `lex`

Unix also provides the `lex` facility. This is a facility that generates programs for the lexical processing of input streams. Lex allows you to pair regular expressions with programs written in C. One reading an input expression, whenever part of it matches against one of the regular expressions, the corresponding program is called.

The general format of a `lex` source file is

Definitions
%%
Rules
%%
User Subroutines

Rules consist of a regular expression on the left hand side, and program fragments on the right hand side e.g.

%%
color printf("colour");

This will print 'colour' whenever input contains 'color' (all other input is copied directly to ouput). The program fragments are written in C, but `lex` makes a few special variables and constructs available

- The string variable *yytext* will contain the input that has been matched by the regular expression
- Variable *yyleng* gives length of matched input
- Default action: copy input to output
- To ignore input, just use statement delimiter, `;`
- To print matched input, call `ECHO`

The user defined subroutines following following the rules provide (a) any extra subroiutines required, and (b) a `main` procedure to tie everything together.

Here for example is a lex version of a rudimentary scanner for pascal-like syntax, which prints out a commentary on what is being scanned:

```
%{
#include <math.h>
#include <stdio.h>
%}
/* definitions for digits and identifiers */
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
%%

/* example of definition expansion */
{DIGIT}+                 {
                          printf("An integer: %s (%d)\n", yytext,
                          atoi(yytext));
                          }
{DIGIT}+"."{DIGIT}*      {
                          printf("A float: %s (%g)\n", yytext,
                          atof(yytext));
                          }
if|then|begin|end|procedure|function        {
                          printf("A keyword: %s\n", yytext);
                          }
{ID}                     printf("An identifier: %s\n", yytext);
"+"|"-"|"*"|"/"          printf("An operator: %s\n", yytext);
"{"[^}\n]*"}"            /* eat up one-line comments */
[ \t\n]+                 /* eat up white space */
.                        printf("Unrecognized char: %s\n", yytext);
%%

int main(int argc, char *argv[])
    {
     ++argv, --argc;  /* skip over program name */
     if (argc > 0)
```

```
   yyin = fopen(argv[0], "r");
else
   yyin = stdin;
yylex();
}
```

Note that this consists of a number of regular expressions paired with program fragments. At the end is the definition of the `main` procedure, which gets hold of its input either from some named file or standard input, and then call the program `yylex`.

The program `yylex` is generated by `lex`. It is basically a collection of finite state machines — one for each regular expression in the lex file. The machines are run in parallel on the input stream. Whenever a machine reaches a final state, the code associated with it is executed.

A major use for `lex` is as a lexical preprocessor for constructing compilers.

3.3 Kleene's Theorem: RegExps ↔ FSAs

We now turn to the equivalence between regular expressions and finite state automata. Kleene's Theorem states that

(a) For every regular expression, there is a finite state machine accepting exactly the same language, and

(b) For every finite state machine, there is a regular expression specifying the language accepted by the machine

First we will show the procedure foring covert FSAs to regular expressions, and regular expressions to FSAs. This is the basis of Kleene's theorem. We will then formally show that these procedures establish the equivalence between FSAs and regular expressions.

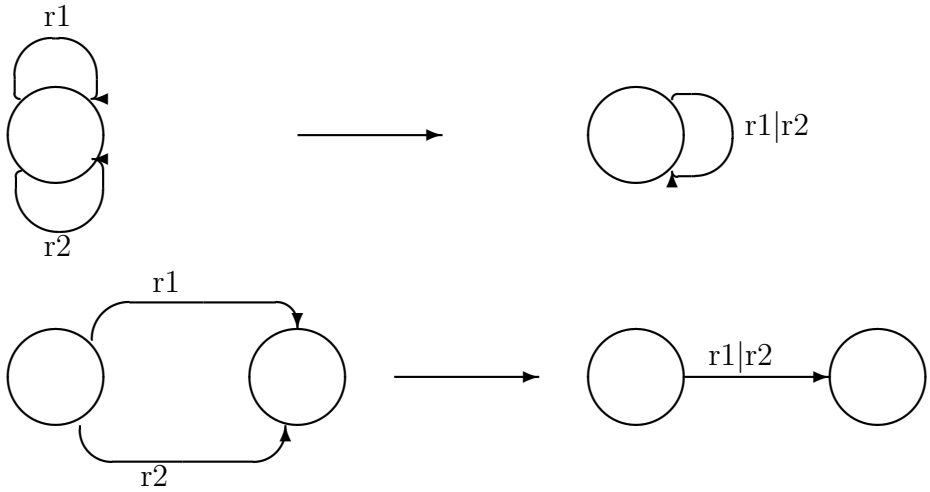
3.3.1 Mapping FSAs to Regular Expressions

The alogorithm for converting a finite state automaton to a regular expression is as follows:

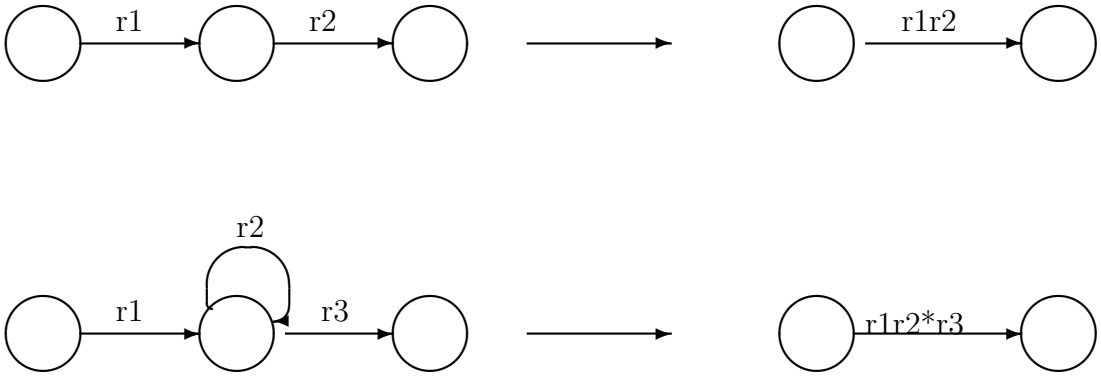
1. Add a new start state to the FSM that has no transitions coming into it
2. Similarly, add a new, unique finish state that has no transitions coming out of it
3. (1) and (2) are acheived by having a single ϵ-transition from the new start state to the old one, and ϵ-transitions from all the old finish states to the new one

4. First, remove multiple arcs connecting the same two states, leaving just one labelled with a regular expression.

   The rules for removing arcs are shown pictorially:



5. Then, pick on a state s_r (other than the start and finish state) to remove. To remove s_r, consider all pairs of states s_i and s_j, and all two transition paths connecting them that pass through s_r. For every such path, add a new arc from s_i directly to s_j, labelled with a regular expression as shown below:



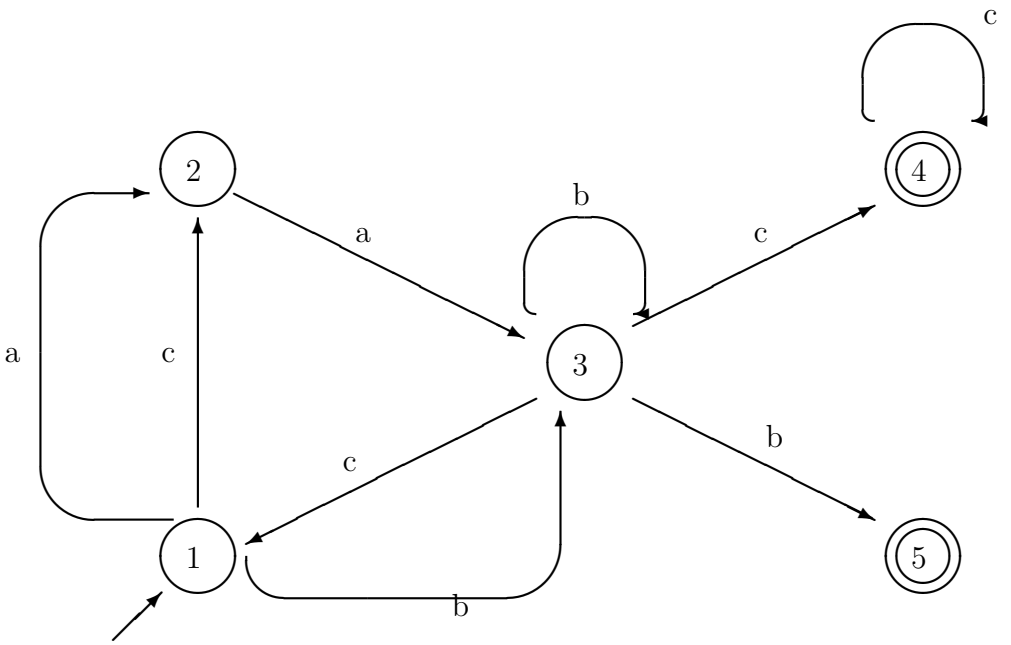
   When new arcs have been added for all paths passing through s_r, the state s_r can be removed along with all arcs into and out of it.

6. Removing a state may have added some new arcs in such a way that there are now multiple arcs directly connecting two states. Therefore, continue performing steps (4) and (5) until there is just one start state, one final state and one arc, labelled with a regular expression.

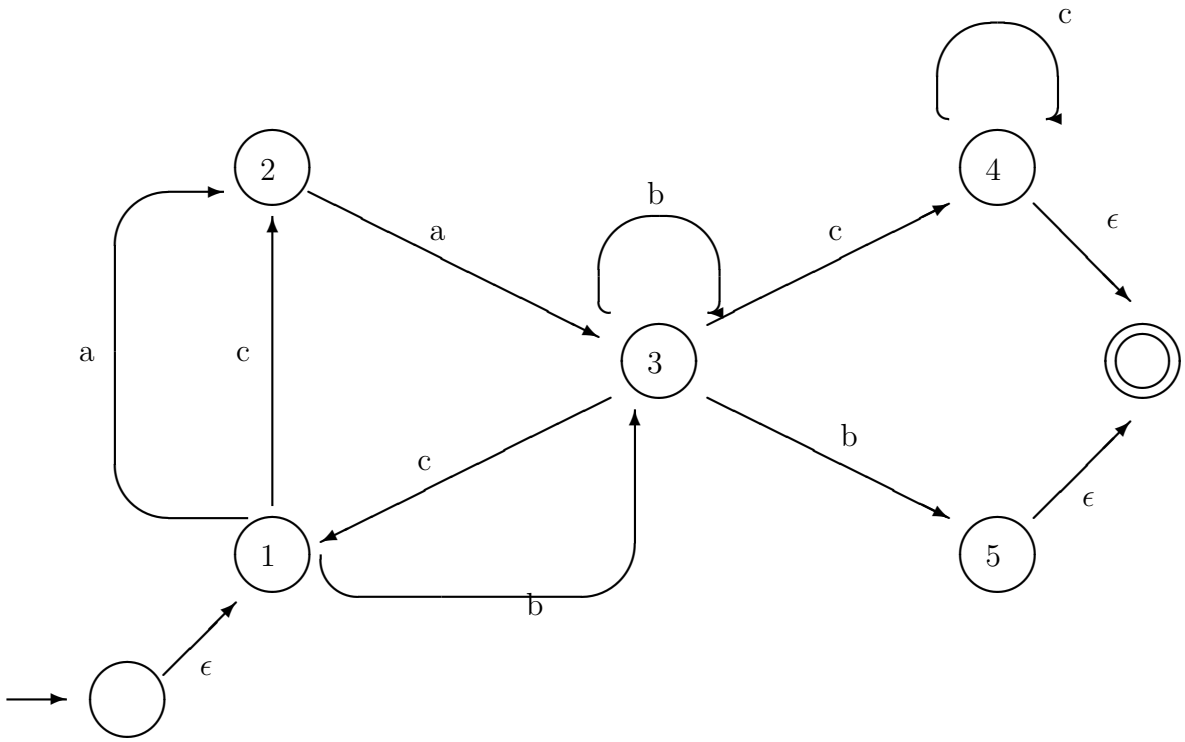
   The regular expression on the final remaining arc defines the same language as the original FSA

Example

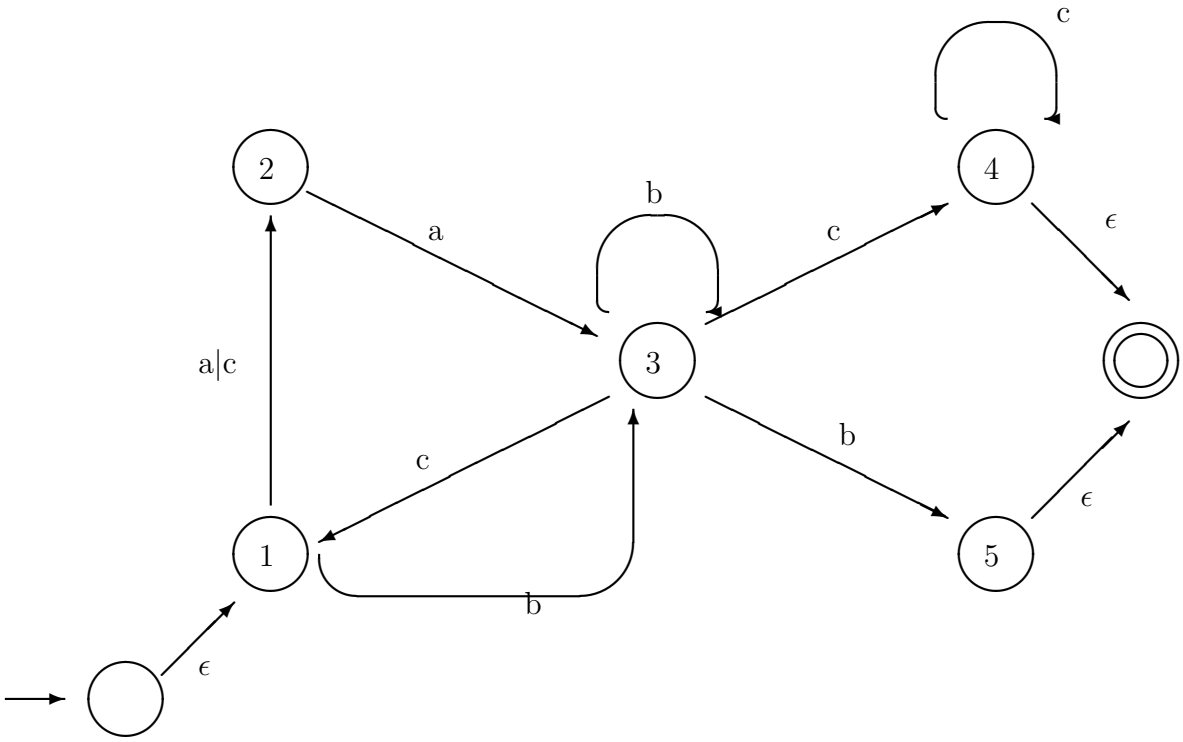
As an example, we will derive a regular expression from the following FSA



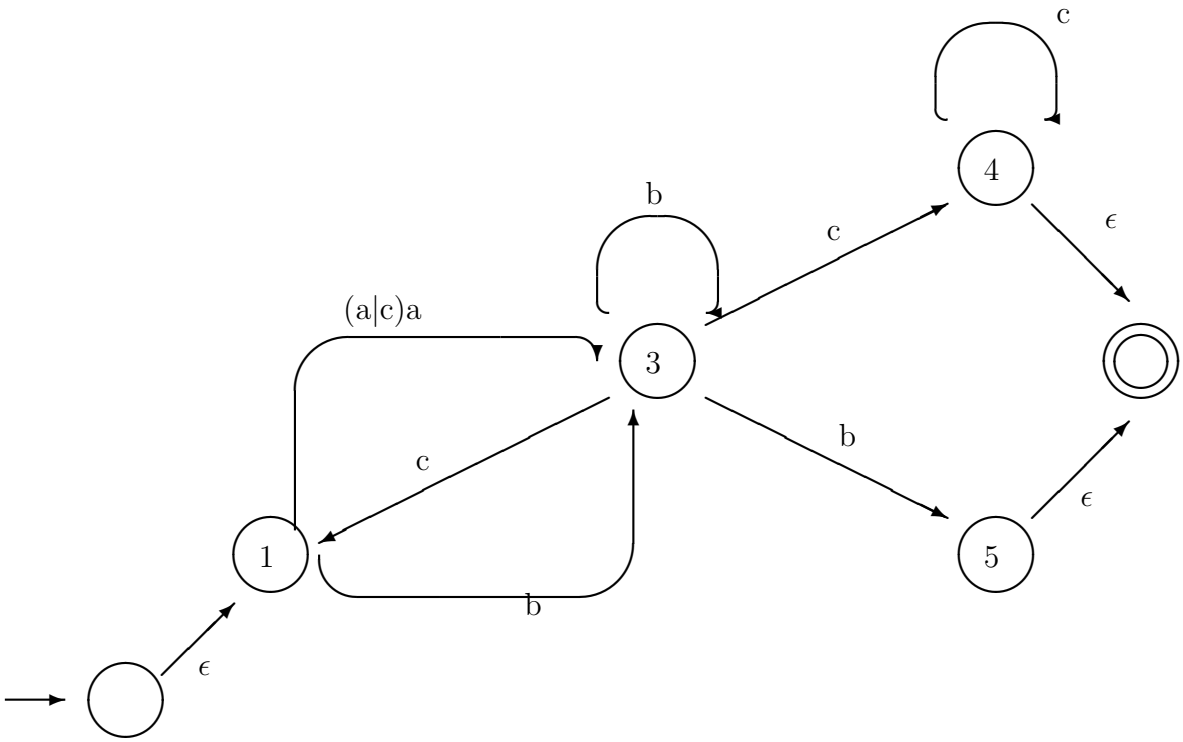
First, we need to add a new start and finish state:



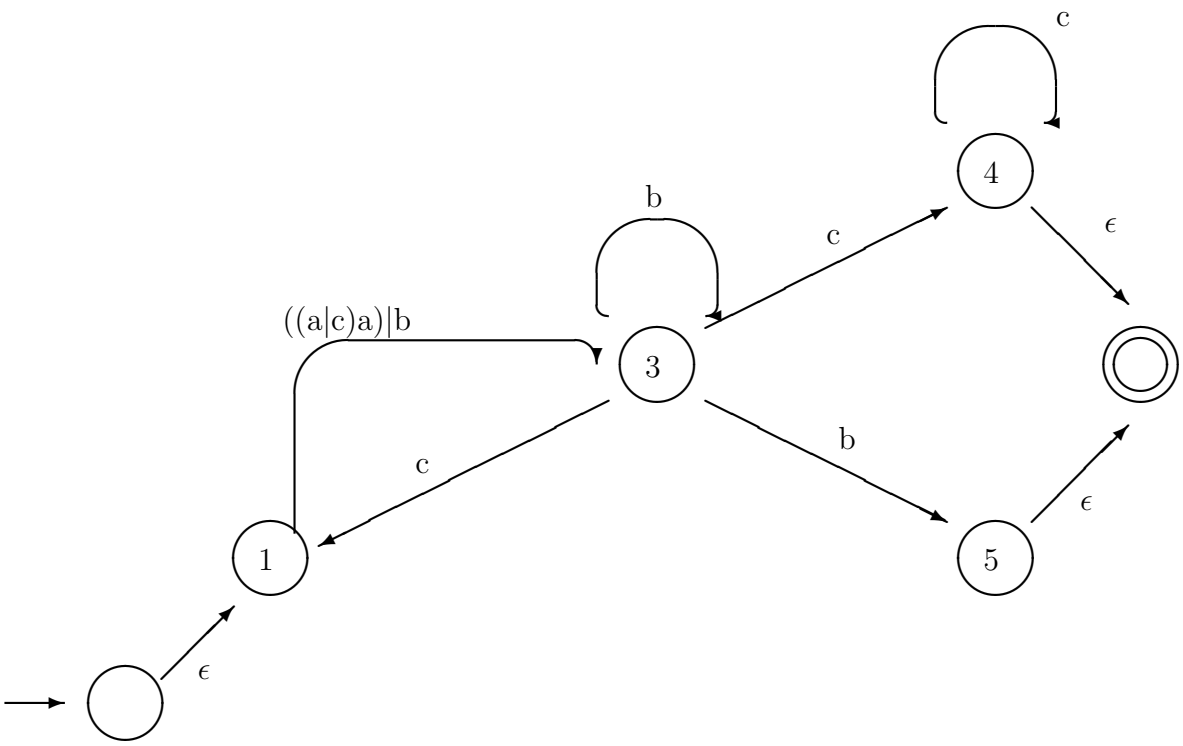
There are two transitions connecting state 1 to state 2, and these can be collapsed into one, as follows:



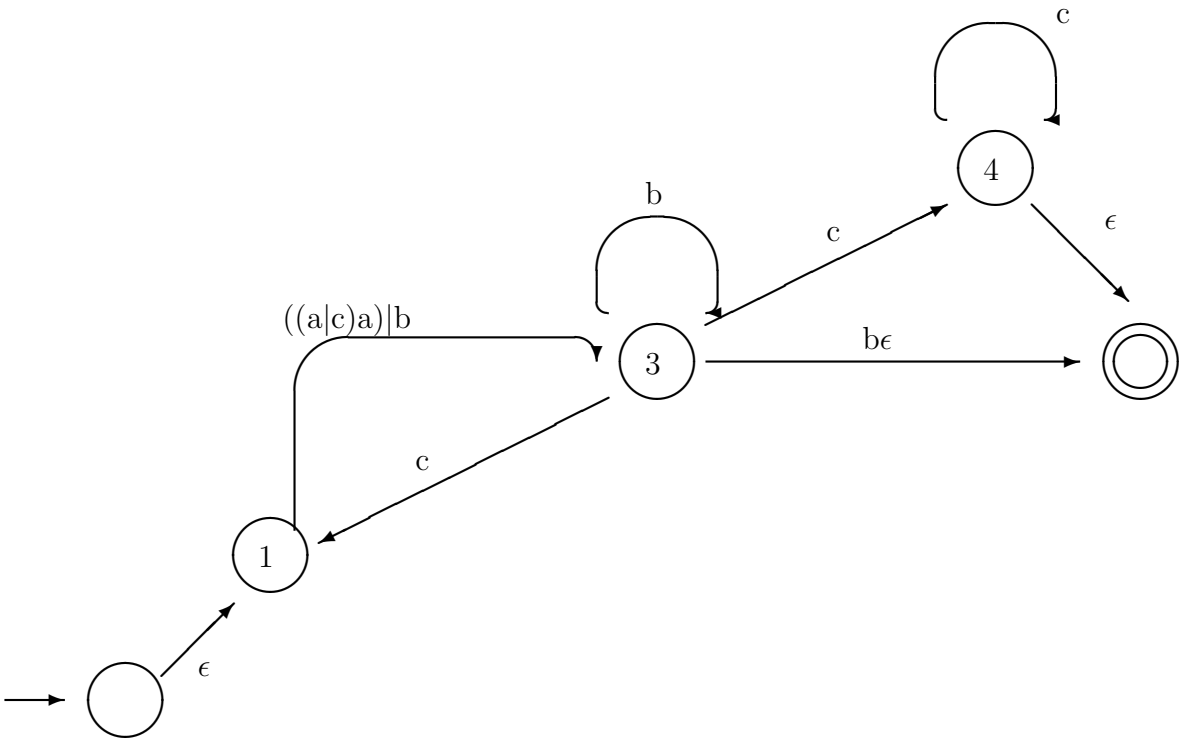
We can now remove state 2. There is just one path passing through it, going from state 1 to state 3. Hence we get:



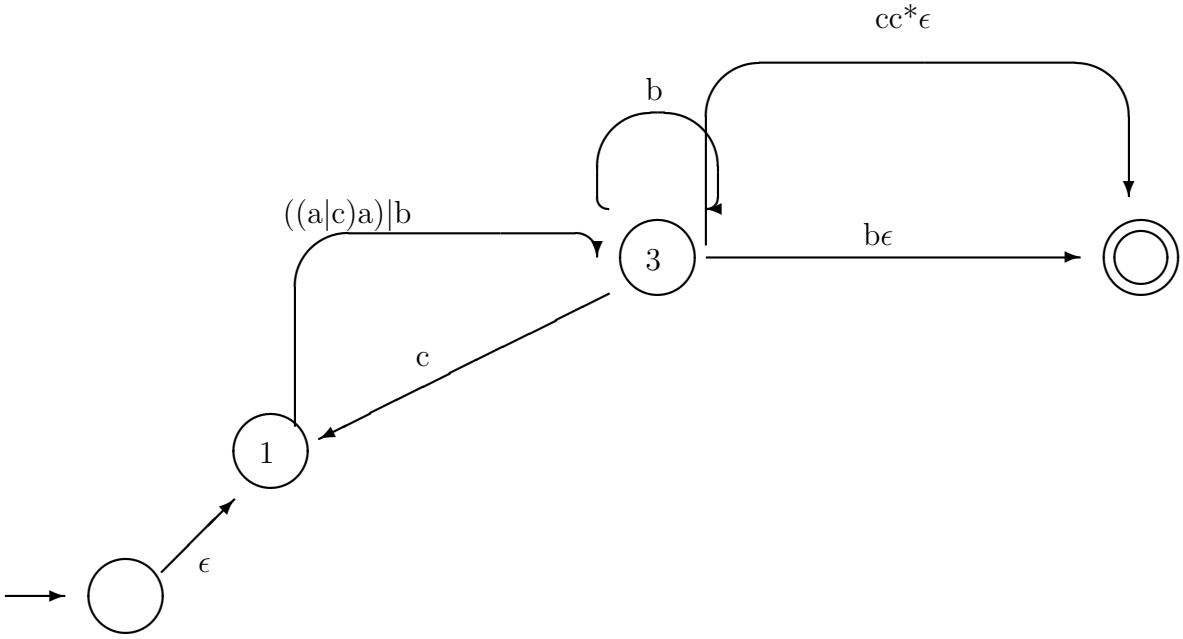
There are now two arcs going from state 1 to state 3, so we collapse these into one:



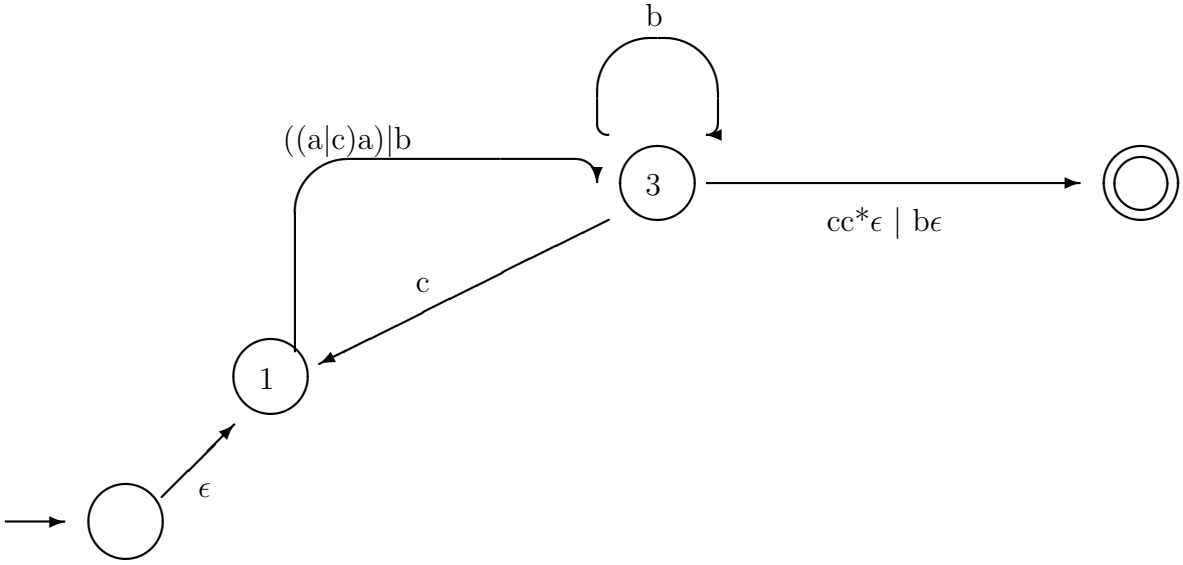
We can now eliminate state 5, which has one path through it from state 3 to the final state:



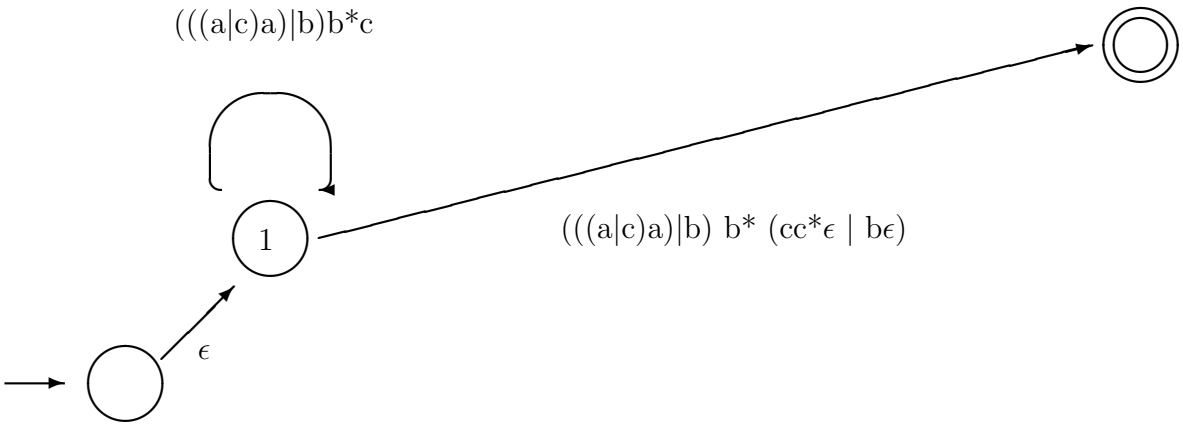
This does not introduce any multiple arcs between states, so we proceed directly to removing state 4. Here there is just one path from state 3 to the final state. But note that there a a transition looping on state 4. Hence we get:



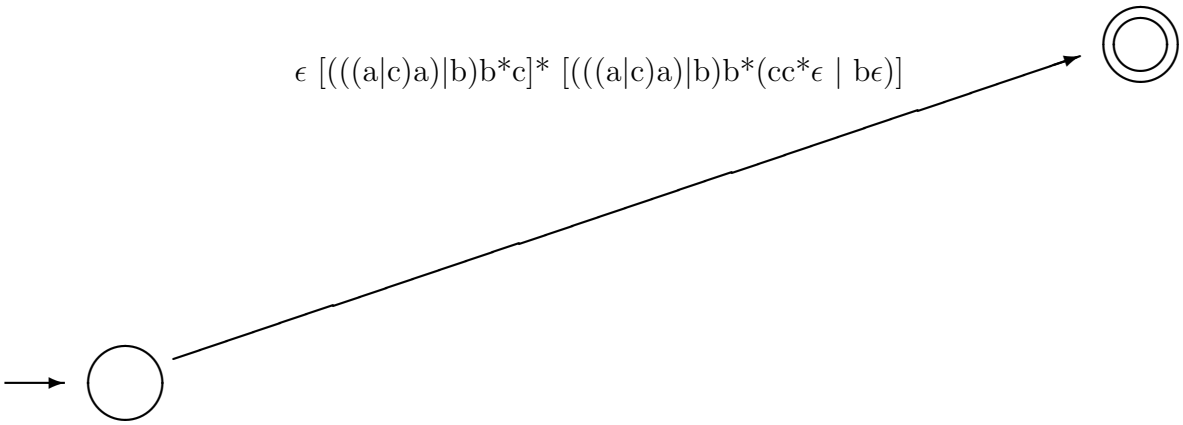
This introduces a second arc going from state 3 to the final state, so we collapse it into one



We can now remove state 3. There is one path going from state 1 back to itself, and one path going from state 1 to the final state. There is also a looping transition on state 3. Thus we add two new arcs:



We can now eliminate state 1, to give a single arc connecting the start and the finish state:



The resulting regular expression is that derived from the machine. Eliminating the ϵ concatenations, we get

[(((a|c)a)|b)b*c]* [(((a|c)a)|b)b*(cc* | b)]

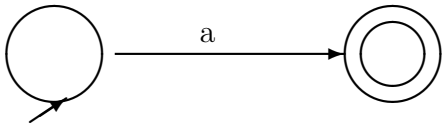
Other orders of removing states would have been possible, and would have led to different but equivalent regular expressions.

3.3.2 Mapping Regular Expressions to FSAs

We now present the procedure for going back in the other direction: constructing a finite state automaton from a regular expression. The idea is to start off with mini-FSAs for each atomic expression in the regular expression. We then define operations for joining together FSAs, one for each of the concatenation, alternation and star operations joining together regular expressions.

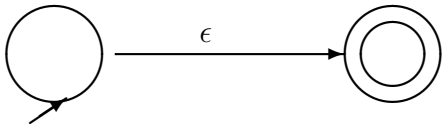
These rules can be represented pictorially as follows:

1. Atomic FSA accepting: **a**

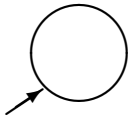


An FSA to accept **a** consists of a single start state and a single final state with an a-transition between them.

2. Atomic FSA accepting: ϵ

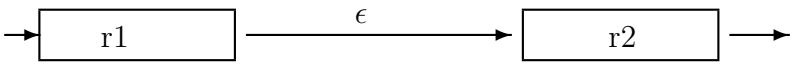


3. Atomic FSA accepting: **0**



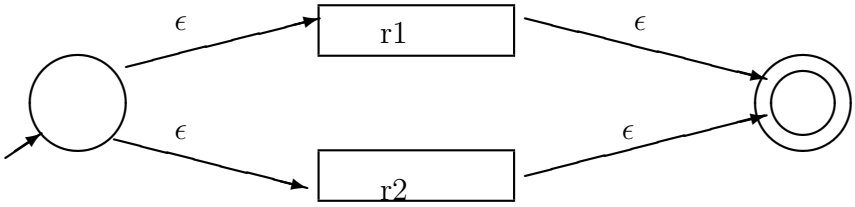
An FSA that accepts nothing at all is just a single start state. (Nb. this rule rarely used).

4. Compound FSA accepting: $\mathbf{r_1 r_2}$



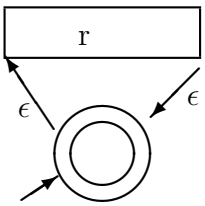
Assuming that we have already constructed an FSA to accept $\mathbf{r_1}$ and one to accept $\mathbf{r_2}$: join the end state of the machine for $\mathbf{r_1}$ to the start state of the machine for $\mathbf{r_2}$ with an empty transition. The start state of $\mathbf{r_1}$ becomes the start state of the compound machine and the end state of $\mathbf{r_2}$ becomes its end state.

5. Compound FSA accepting: $\mathbf{r_1} \mid \mathbf{r_2}$



Assuming machines for $\mathbf{r_1}$ and $\mathbf{r_2}$: create a new start state with empty transitions to start states of $\mathbf{r_1}$ and $\mathbf{r_2}$, and e new final state with empty transitions from the final states of $\mathbf{r_1}$ and $\mathbf{r_2}$.

6. Compound FSA accepting: $\mathbf{r}^*$



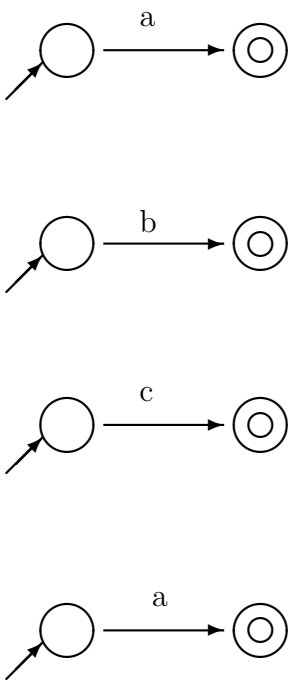
Given a machine for $\mathbf{r}$: Create a new state that is the start and finish state of the compound machine, and add an empty transition from it to the start of $\mathbf{r}$, and and empty transition from the final state of $\mathbf{r}$ to it.

Example

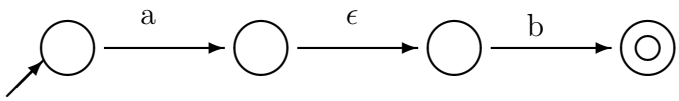
To see how these rules work, we will construct an FSA from the regular expression

$$(\mathbf{ab} \mid \mathbf{c}^*)^*\mathbf{a}$$

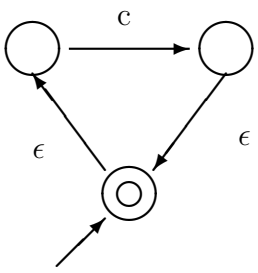
First we construct FSAs for the atomic components, $\mathbf{a}, \mathbf{b}, \mathbf{c}$ and $\mathbf{a}$:



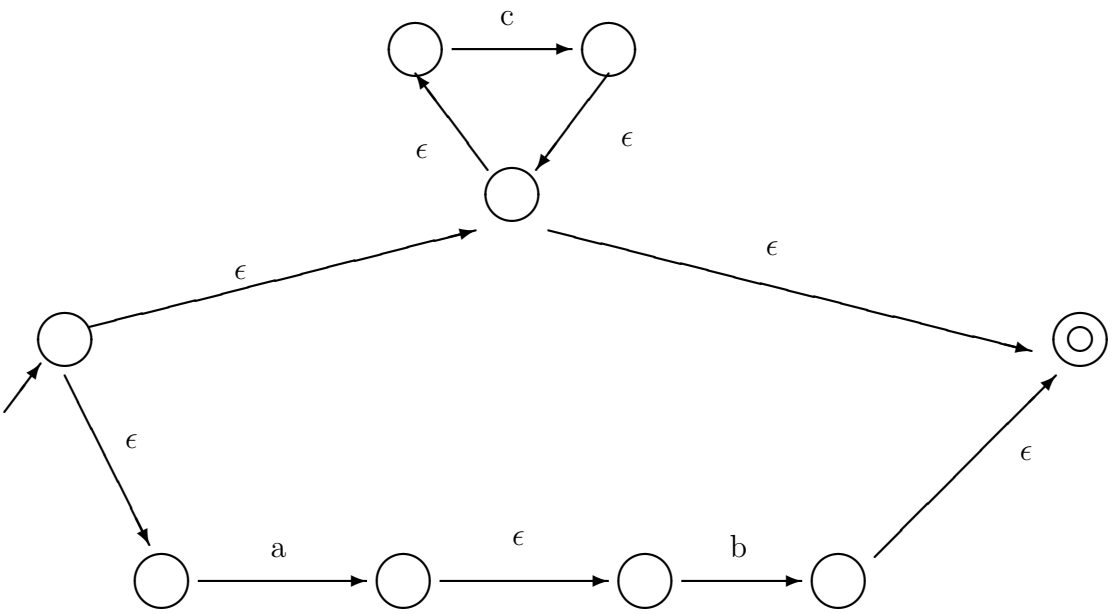
To form the FSA corresponding to the regular expression **ab**, we join the end state of **a**'s machine to the start state of **b**'s machine, using an empty transition:



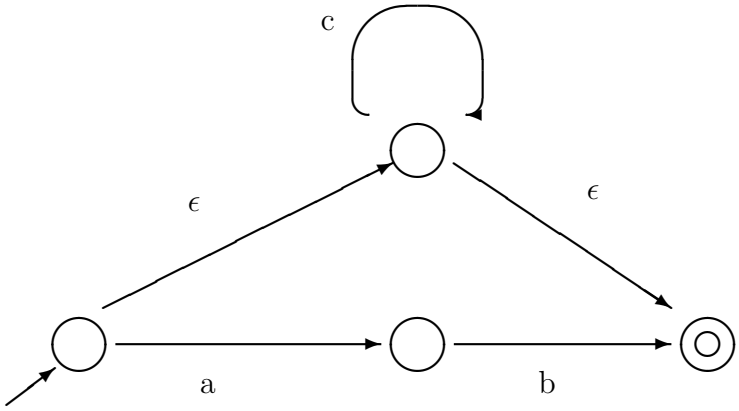
We also construct a machine for $\mathbf{c}^*$ from that for **c**



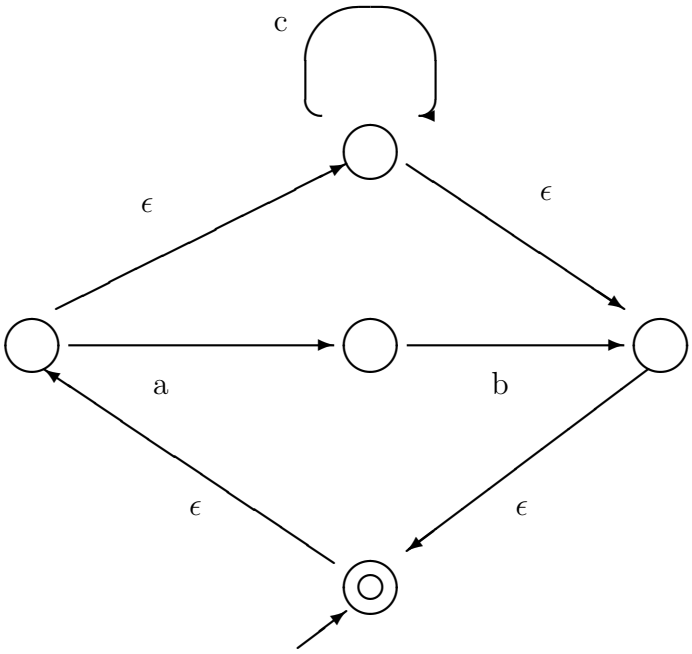
We then join these two machines together, according to the rule for alternation:



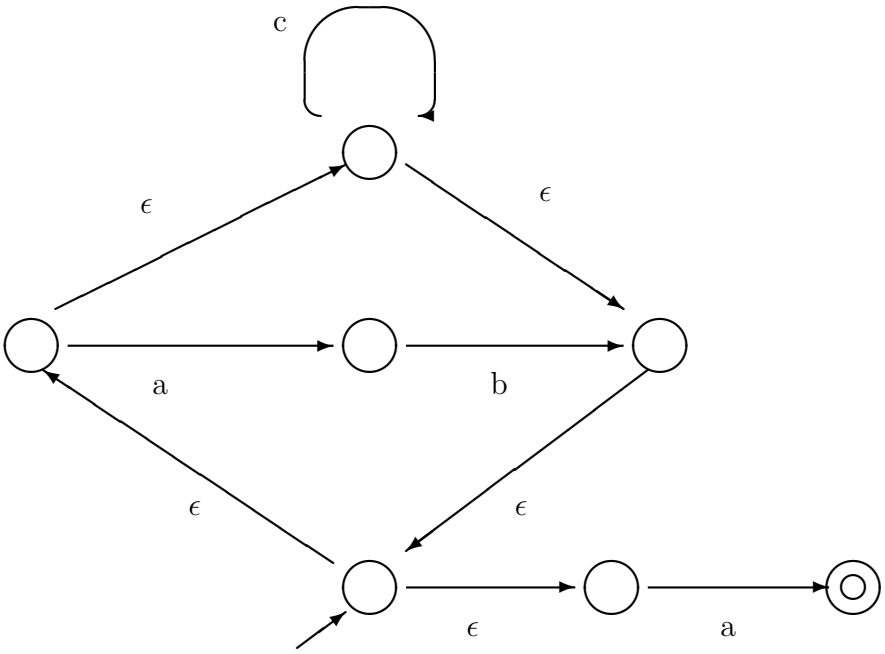
To keep things simple, we can remove some of the excess empty transitions, to get the following machine also equivalent to **ab** | **c***:



We now have to apply the Kleene star to this machine, to get:



We finish by concatenating this to the machine for **a**:



To recapitulate: to build an FSA from a regular expression, we start by building the atomic FSAs corresponding the atomic expressions in the regular expression. We then build the machine up from it atomic machines, in the same order that the regular expression is built up from its atomic expressions, using the rules shown previously.

3.3.3 Formal Proofs of Kleene's Theorem

The last two sections showed how to convert between FSAs and regular expressions, but without proving that the conversion preserves the language being defined. We now turn to this.

a) From FSAs to RegExps

Suppose we start with machine M:

$$M = \langle S, V, \delta, s_0, F \rangle$$

where $S = \{s_0, s_1, \ldots, s_n\}$. That is, the states have an arbitrary (numerical) ordering imposed on them.

Define a set of strings, L_{ij}^k, taking you from state s_i to state s_j, and passing only through intermediate states s_l where $l \leq k$

> $w \in L_{ij}^k$ iff
> $s_j = \delta^*(s_i, w)$,
> and for all proper prefixes, u, of w,
> $\delta^*(s_i, u) \in \{s_l \mid l \leq k\}$
> For $0 \leq k \leq n$, $1 \leq i, j \leq n$

Claim: L_{ij}^k is denoted by a regular expression for all i, j

Proof by induction on k

Base case $k = 0$ (a) No arcs connecting s_i directly to s_j:
L_{ij}^0 is empty, hence re=$\mathbf{0}$

(b) Arcs connecting s_1 directly to s_j:
L_{ij}^0 is a set of atomic words, $\{a_1, a_2, \ldots, a_r\}$,
hence re = $\mathbf{a_1} \mid \mathbf{a_2} \mid \ldots \mid \mathbf{a_r}$

Induction Assume L_{ij}^k is denoted by an re for all i, j, and show the same is true for $k + 1$

Note that

$$L_{i,j}^{k+1} = L_{i,j}^k \cup L_{i,(k+1)}^k (L_{(k+1),(k+1)}^k)^* L_{(k+1),j}^k$$

(Strings from i to j not passing through $k+1$, plus strings from i to $k+1$ concatenated with strings from $k+1$ to itself, finished off by strings from $k+1$ to j.)

By induction hypothesis, all sub-terms are regular expressions, and hence $L_{i,j}^{k+1}$ is a regular expression.

Given the proof of this claim, we now reason as follows. Suppose the final accepting states in the machine M are

$$F = \{s_{n_1}, \ldots s_{n_m}\}$$

Then the language accepted by M is

$$L(M) = L^n_{0,n_1} \cup \ldots \cup L^n_{0,n_m}$$

This corresponds to a disjunction of regular expressions, which is itself a regular expression. So for any finite state automaton, a corresponding regular expression exists.

b) From RegEexps to FSAs

The proof is by induction on the structure of the regular expression. The base case involves constructing FSAs for atomic regular expressions $(\epsilon, \epsilon, \mathbf{a})$. The induction shows how to combine FSAs for sub-expressions in alternations (|), concatenations, and Kleene star (*).

Base Case: Construct FSAs for null language and atomic sentences as shown in section 3.3.2.

Recursion

1. Alternation (|):
   Given FSAs M_1 and M_2, construct NFA N with $L(N) = L(M_1) \cup L(M_2)$:

   $$M_1 = \langle S_1, V, \delta_1, s_{01}, F_1 \rangle \; M_2 = \langle S_2, V, \delta_2, s_{02}, F_2 \rangle \; N = \langle S, V, \delta, s_0, F \rangle$$

   where
   $S = S_1 \times S_2$
   $\delta(\langle s_1, s_2 \rangle, a) = \langle \delta_1(s_1, a), \delta_2(s_2, a) \rangle$
   $s_0 = \langle s_{01}, s_{02} \rangle$
   $F = \{\langle s_1, s_2 \rangle \mid s_1 \in F_1 \text{ or } s_2 \in F_2\}$

   Note that:
   $w \in L(N)$ iff $\delta(\langle s_{01}, s_{02} \rangle, w) \in F$
   iff $\delta_1(s_{01}, w) \in F_1$ or $\delta_2(s_{02}, w) \in F_2$
   iff $w \in L(M_1) \cup L(M_2)$

2. Concatenation:
   Given DFAs M_1 and M_2, construct NFA N with $L(N) = L(M_1)L(M_2)$:

   $$N = \langle S, V, \delta, s_0, F \rangle$$

   where
   $S = S_1 \cup S_2$
   $s_0 = s_{01}$
   $F = F_2$ (if $s_{02} \notin F_2$), otherwise $F_1 \cup F_2$
   $\langle s, a, s' \rangle \in \delta$ iff
   (i) $s \in S_1$ and $s' = \delta_1(s, a)$, or
   (ii) $s \in S_2$ and $s' = \delta_2(s, a)$, or
   (iii) $s \in F_1$ and $s' = \delta_2(s_{02}, a)$

   We need to prove that $L(M_1)L(M_2) \subseteq L(N)$:

   Let $w = w_1 w_2$, where $w_1 \in L(M_1), w_2 \in L(M_2)$

Case 1 $w_2 = \epsilon$ (i.e. $\epsilon \in L(M_2)$)

Since $\epsilon \in L(M_2)$, $s_{02} \in F_2$, hence $F = F_1 \cup F_2$

Also $\delta_1^*(s_{01}, w_1) \in F_1 \subseteq F$

Hence $w \in L(N)$

Case 2 $w_2 = aw_2^{'}$

Let $\delta_1^*(s_{01}, w_1) = s_{1f} \subseteq F_1$

There is a type (iii) transition, $\delta_2(s_{02}, a) = s_{2a}$

Followed by a type (ii) transition, $\delta_2^*(s_{2a}, w_2^{'})$

Combining these two, $\delta_2^*(s_{02}, aw_2^{'}) \in F_2 \subseteq F$

Hence $w \in L(N)$

Proof that $L(N) \subseteq L(M_1)L(M_2)$ is similar

3. Kleene star:
Given FSA M, construct a NFA, N with $L(N) = L(M)^*$:

$$M = \langle S, V, \delta, s_0, F\rangle \; N = \langle S', V, \delta', s_0', F'\rangle$$

where

$s_0' =$	an entirely new state
$S' =$	$S \cup \{s_0'\}$
$F' =$	$F \cup \{s_0'\}$
$\delta'(s, a, s')$	iff
	(i) $s \in S$ and $s' = \delta(s, a)$, or
	(ii) $s \in F$ and $s' = \delta(s_0, a)$, or
	(iii) $s = s_0'$ and $s' = \delta(s_0, a)$

Suppose $w \in L(N)$, so that $\delta^{'*}(s_0', w) = q \in F' = F \cup \{s_0'\}$

Suppose $q = s_0'$: it must be that $w = \epsilon$, hence $w \in L(M)^*$

Suppose $q \in F$: The will be a sequence of transitions

type (iii)(type(i) type(ii))*

such that $w = a_1w_1a_2w_2 \ldots a_nw_n$, where a_1 give the initial type(iii) transition, and a_j the type(ii) transitions.

The states resulting after the type(ii), a_j, transitions are by definition $\delta(s_0, a_j)$.

Hence $\delta(s_0, a_jw_j) \in F$, i.e. $a_jw_j \in L(M)$

Hence $(a_1w1)(a_2w_2) \ldots (a_nw_n) \in L(M)^*$

3.4 Regular Grammars

As Kleene's theorem shows, regular expressions are one way of defining the kind of languages accepted by finite state automata. Another way of defining these languages is by means of a regular grammar.

A grammar is a tuple

$$G = \langle N, V, P, S \rangle$$

where
N is a set of non-terminal symbols
V is a set of terminal symbols (the vocabulary)
P is a set of productions or rules
$S \in N$ is a priviliged non-terminal start symbol

Typically, the terminal and non-terminal symbols are disjoint: $N \cap V = \{\}$
Rules take the form

$$\text{LHS} \rightarrow \text{LHS}$$

where
LHS is a sequence of terminals and/or non-terminals

RHS is a (possibly empty) sequence of terminals and/or non-terminals

This definition of a grammar covers all the types of grammar that we will encounter on this course. Different types of grammar are obtained by imposing different restrictions on what count as allowable grammar rules. For regular grammars, the restriction on the form of the rules is:

Regular Grammar

This is a grammar where
(a) the LHS must be a single non-terminal
(b) the RHS must either be
  (i) a single terminal, or
  (ii) a single non-terminal followed by single terminal.

This definition describes a *left linear regular grammar*. An alternative and equally good definition is

(Right Linear) Regular Grammar

This is a grammar where
(a) the LHS must be a single non-terminal
(b) the RHS must either be
  (i) a single terminal, or
  (ii) a single terminal followed by single non-terminal.

Right linear grammars define the same range of languages as left-linear regular grammars. However, it is important to note that you *cannot mix left- and right-linear rules in the same grammar*, and still get a regular grammar.

An example of a right linear grammar is

S1 → a S2	S1 → the S2
S2 → cat S3	S2 → dog S3
S3 → saw S4	S3 → likes S4
S4 → a S5	S4 → the S5
S5 → cat	S5 → dog

An equivalent left linear grammar is

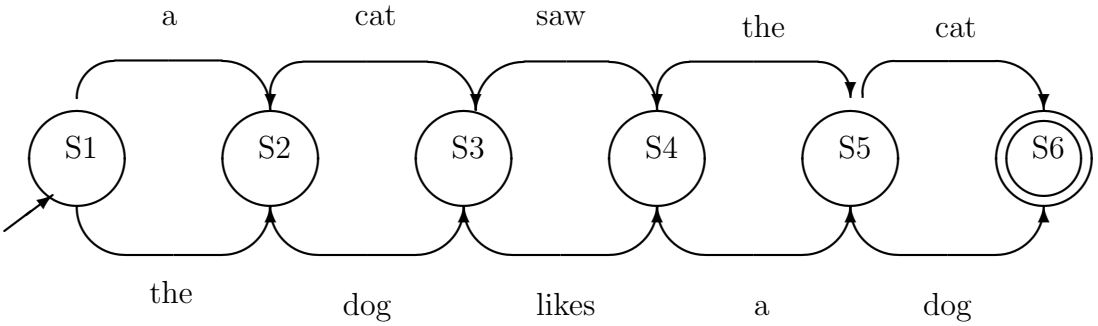
S1 → S2 a	S1 → S2 the
S2 → S3 cat	S2 → S3 dog
S3 → S4 saw	S3 → S4 likes
S4 → S5 a	S4 → S5 the
S5 → cat	S5 → dog

3.5 Kleene's Theorem: RegGrams ↔ FSAs

We can extend Kleene's theorem, which makes the connection between finite state automata and regular expressions to include regular grammars.

> Any regular grammar can be converted to a finite state machine, and vice versa

Example:



Grammar:

S1 → a S2	S1 → the S2
S2 → cat S3	S2 → dog S3
S3 → saw S4	S3 → likes S4
S4 → a S5	S4 → the S5
S5 → cat	S5 → dog

In outline form, the conversion is:

- States becomes non-terminals
- Label on transition plus result state becomes RHS of rule rewriting initial state
- Except for transitions to final states (e.g. S6), where the RHS is just the label on the transition.

Consequently, regular grammars generate/accept the same class of languages as regular expressions

3.6 Summary

Languages defined by finite state automata are known as *regular languages*. In this chapter, we introduced two further ways of defining regular languages:

- Regular expressions
- Regular grammars

Kleene's theorem shows how any FSA can be converted to a regular expression, and how any regular expression can be converted to an FSA. We also showed how FSAs (without ϵ-transitions) can be converted to regular grammars, and vice versa.

Not all formal languages are regular. In the next chapter, we establish some limits on what languages are and are not regular. That is, we establish limits on what languages can be defined by any (or all) of: finite state automata, regular expressions, regular grammars

Chapter 4

Closure Properties of Regular Languages

Not all formal languages are regular. That is, not all formal languages can be defined by either a finite state automaton, a regular expression, or a regular grammar. This chapter first looks at a way of determining whether a given language is regular or not, known as the *pumping lemma*. It then looks at various closure properties of regular languages: how one regular lanaguage may be combined with another to give something that is still regular.

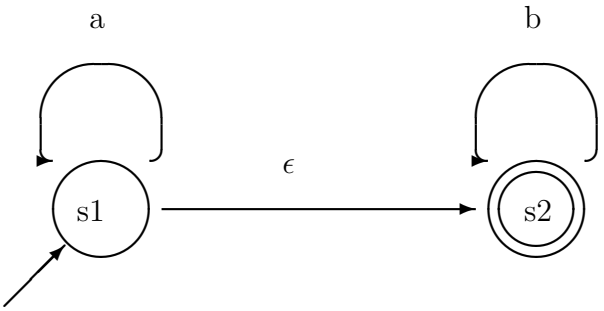
In the next part of these notes, we will look at context free languages, which are defined by automata somewhat more elaborate than finite state machines, and grammars somewhat less restricted than regular grammars.

4.1 Pumping Lemma for Regular Languages

4.1.1 FSAs Have no Memory

Before presenting the pumping lemma for regular languages, we will try to get a grip on the intuition behind it. This is as follows. Finite state automata have no memory. What transition you take is determined purely by the current state and the current word of input. No account is taken of how you arrived at that state to begin with.

Consider an FSA accepting the language $a^n b^m$ (that is, n occurrences of the letter a, followed by m occurences of the letter b, for any n and m)



When you enter state $s2$, there is no way of recording how many a-transitions looping on state $s1$ were taken beforehand. Therefore, there is no way of limiting the number b-transitions looping on state $s2$ according to the number of a-transitions taken.

This means that the language $a^n b^n$ (any number of as followed by the same number of bs) cannot be defined by a finite state machine. Although the machine above accepts strings of the form $a^n b^n$, it does not accept *only* strings of this form. To do so, it would need to be able to keep track of the number of a-transitions taken, and then take exactly the same number of b transitions. That is, an automaton defining a language like $a^n b^n$ requires a certain amount of memory.

The pumping lemma is a way of determining which languages require a degree of memory in their automata.

4.1.2 Statement of Pumping Lemma

The pumping lemma for regular languages may be stated as follows.

Pumping Lemma for Regular Languages:
For every regular language L, there is a number $k \geq 1$ such that all strings $w \in L$ of length k or more can be expressed as

$$w = u_1 v u_2$$

where u_1, v and u_2 satisfy

- length$(v) \geq 1$
- length$(u_1 v) \leq k$
- for all $n \geq 0$, $u_1 v^n u_2 \in L$

What does this all mean?

First note that all finite state machines have a finite number of states. Suppose that a given machine M has k states. Consider a string accepted by M of length

$> k$. To accept this string we have to make transitions from the start state to a final state, and each transition accounts for one word in the string. Since there are more words in the string than there are states in the machine, this means that we must have passed through at least one state more than once. Or in other words, in accepting the string there is at least one extended loop that starts in some state s' and ends in it.

We can therefore break the string into three parts. The first part, u_1 takes you from the start state to state s'. The second part, v, takes you from state s' back to s'. And the final part of the string, u_2, takes you from s' to a final state.

Now, we could have got from the start to the finish state of M without looping round on state s': i.e. we could have missed out the v part of the string, so that u_1u_2 is accepted by M. Likewise, we could also go round the loop any number of times, so that $u_1v^2u_2$, $u_1v^3u_2$, $u_1v^4u_2$, etc., are all accepted by M.

In short, finite state machines have no memory. Thus if a machine allows loops, we cannot place any restrictions on the number of times the loop may be taken. If a machine with k states can accept strings u_1vu_2 of length greater than k, then there must be some part of the string, v, corresponding to a loop. This looping part can be omitted, or repeated as often as one likes. Hence strings $u_1v^nu_2$ $(n \geq 0)$ will also be accepted by the machine.

4.1.3 Negative Form of Pumping Lemma

The pumping lemma for regular languages is normally used in its negative form. Instead of showing that some language is regular, we use it to show that some language *cannot* be regular. To do this we first identify some portion of a string corresponding to a loop, and then show that there are restrictions on the number of times that the loop can be repeated. The imposition of these restrictions requires some form of memory, meaning that the machine cannot be finite state.

The negative form of the pumping lemma is

> *(Negative) Pumping Lemma for Regular Languages*:
>
> To show that some language L is *not* regular:
>
> Find one string $w \in L$ of length greater than k,
>
> such that no matter how you decompose w into u_1vu_2
>
> (where length$(u_1v) \leq k$ and length$(v) \geq 1$)
>
> there is some $n \geq 0$ such that $u_1v^nu_2 \notin L$

That is, to show that a language is not regular, we have to find just one string in the language over a certain length (k), where no matter how it is decomposed into three parts (u_1, v and u_2), there are restrictions on the number of times that the v part can be repeated.

The one problem with this is that we cannot tell what value should be chosen for k. (It is the number of the states in the putative FSA for the language, but we only have access to the language and not its automaton.) We therefore have to use strings parametrized in some way on k. The following tow examples should illustrate this.

4.1.4 Examples of Using Pumping Lemma

We give two examples showing how the negative version of the pumping lemma is used.

$a^n b^n$ is not Regular

To show that $a^n b^n$ is not regular, consider some string $a^k b^k$. We know that this string is longer than k, whatever the value of k happens to be.

Now consider all ways of decomposing the string such that

$$a^k b^k = u_1 v u_2$$

(where $\text{length}(u_1 v) \leq k$ and $\text{length}(v) \geq 1$).

In fact, there is just one general decomposition:

$$a^k b^k = a^{r-s}\ a^s\ a^{k-r} b^k$$

where $s \geq 1$ and $r \leq k$. i.e.

- $u_1 = a^{r-s}$
- $v = a^s$
- $u_2 = a^{k-r} b^k$

Now consider $n = 0$:

$$\begin{aligned} u_1 v^0 u_2 &= u_1 u_2 \\ &= a^{r-s} a^{k-r} b^k \\ &= a^{k-s} b^k \end{aligned}$$

But $a^{k-s} b^k$ is not in $\{a^n b^n \mid n \geq 1\}$, so the language is not regular.

$\{a^p \mid p$ is prime$\}$ is not Regular

This is a rather more complex application of the pumping lemma (more involved than you would find in an exam question). We can show that the language consisting of the strings a^p, where p is a prime number, is not regular.

For some $p \geq k$ let

$$w = a^p = u_1 v u_2$$

where

- $u_1 = a^{r-s}$
- $v = a^s$
- $u_2 = a^{p-r}$
- and $s \geq 1$ and $r \leq k$

Now consider the string $u_1 v^{p-s} u_2$

$$u_1 v^{p-s} u_2 = a^{r-s} a^{s(p-s)} a^{p-r} = a^{(s+1)(p-s)}$$

But $(s+1)(p-s)$ is not prime (it has at least two factors). So, for $n = p - s$, $u_1 v^n u_2 \notin L$. Hence the language is not regular.

4.1.5 Proof of Pumping Lemma

The formal proof of the pumping lemma for regular languages is as follows.

Take some regular language, L

1. Since L is regular, let $L = L(M)$ for some DFA $M = \langle s, V, \delta, s_0, F \rangle$
2. Let k be the number of states in M
3. Given a string $w = a_1 a_2 \dots a_m$, $m \geq k$, put

   $q_0 = s_0$
   $q_1 = \delta(q_0, a_1) \ \dots \ q_m = \delta(q_0, a_1 a_2 \dots a_m) = \delta^*(q_0, w) \in F$
4. Since there are only k states, $k \leq m$, the $k+1$ states $q_0 \dots q_k$ cannot all be distinct
5. Let j be the least number for which there is some $i < j$ with $q_i = q_j$
6. Clearly, $1 \leq j \leq k$
7. Put
   - $u_1 = a_1 \dots a_i$
   - $v = a_{i+1} \dots a_j$
   - $u_2 = a_{j+1} \dots a_k$
8. v causes a transition from q_i to itself, and hence so does v^n for any $n \geq 0$
9. Thus $\delta^*(q_0, u_1 v^n u_2) = \delta^*(q_i, v^n u_2) = q_k \in F$
10. Hence $u_1 v^n u_2 \in L$ for all $n \geq 0$
11. Since $i < j$, the length of v is at least 1
12. And the length of $u_1 v$ is j, $j \leq k$ by construction

4.2 Closure Properties of Regular Languages

4.2.1 Closure Properties

The following are some closure properties of regular languages:

1. Any finite language is regular
2. The union of two regular languages is regular
3. The concatenation of two regular languages is regular
4. The Kleene closure of a regular language is regular
5. The intersection of two regular languages is regular
6. The complement of a regular language (i.e. all the strings not accepted by the language) is regular

Facts (1)–(4) follow directly from construction of regular expressions, given that all regular languages have corresponding regular expressions.

1. For any finite language, we can explicitly enumerate the strings of the language. Each string can be represented as a regular expression, and the entire language as a regular expression consisting of a finite disjunction of these strings:
   $\mathbf{u_1} \mid \mathbf{u_2} \mid \ldots \mid \mathbf{u_n}$
2. Each language has a corresponding regular expression ($\mathbf{r_1}$ and $\mathbf{r_2}$). The union of the two languages is represented by the regular expression $\mathbf{r_1} \mid \mathbf{r_2}$. Since this is a regexp, it too represents a regular language
3. Similarly, we can represent the concatenation of two regular languages as the concatentation of their regular expressions, $\mathbf{r_1 r_2}$, which in turn represents a regular language.
4. Again, we represent the Kleene closure of a language as the Kleene star of its regular expression, $\mathbf{r}^*$

The remaining two properties, closure under intersection and complement, are proved by the construction of FSA, rather than regular expressions.

5 To show that the complement of a regular language L is regular, consider the FSA M such that $L = L(M)$ (and where M contains an error state).

Form the complement of M, $M' = \langle S, V, \delta, s_0, F' \rangle$, by setting

$$F' = S - F$$

i.e. swap final and non-final states around, but keep everthing else the same.

Everything that M rejects, i.e. that leads to a non-final state in M, will lead to a final state in M'

Everything that M accepts, i.e. that leads to a final state in M, will lead to a non-final state in M'

[Question: why must M include an error state?]

6 To show the intersection of two regular languages is regular, note that $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$

I.e. Take the complements of L_1 and L_2 (which are regular)

And then take the union of the complements, which is also regular.

And then take the complement of this, which is also regular

4.2.2 Equivalence between FSAs

How can we tell whether two FSAs accept the same language? If two FSAs are equivalent (accept the same language), then the intersection of the language of one with the complement of the other will be empty. That is

$L_1 \cap \overline{L_2} = \{\}$, and
$L_2 \cap \overline{L_1} = \{\}$

Hence $(L_1 \cap \overline{L_2}) \cup (L_2 \cap \overline{L_1}) = \{\}$

So construct a machine that accepts

$$(L_1 \cap \overline{L_2}) \cup (L_2 \cap \overline{L_1})$$

This is done as follows:

- Note that $L_1 \cap \overline{L_2} = \overline{L_1} \cup \overline{\overline{L_2}} = \overline{L_1} \cup L_2$
- So we can construct the machine by forming complements and unions of the two machines, which we already know how to do

Having constructed this machine, we no need to see whether it accepts any sentences.

FSA Colouring

An FSA accepts a string if there is some path from the start state to the end state. The following is an *effective procedure* for determining whether there is such a path (an effective procedure is one that produces a result in a finite number of steps)

1. Mark the start state, e.g. by colouring it blue
2. From every marked state, follow each edge out of it to the next state, mark the next state, and delete the edge
3. Repeat step 2 until no more new states are marked
4. Look to see if one of the final states is marked (If not, then the FSA accepts no strings)

Note that step (2) cannot be repeated more then k times, where k is the number of states in the machine.

We can thus tell if two FSAs are equivalent by (a) forming a new FSA that is the intersection of one with the complement of the other, and (b) applying the FSA colouring algorithm to see whether this machine accepts any strings.

Part II

Context Free Languages

The following four chapters introduce the next level up in the Chomsky hierarchy: context free languages. In chapter 5 we will cover how context free languages are defined in terms of context free grammars. In chapter 6 we will look at how context free languages are defined by a particular kind of automaton known as a pushdown stack automaton (basically a finite state machine with an additional stack-like memory). Chapter 7 covers the topic of parsing with context free grammars, which is particularly important for compiling (context free) programming languages. Chapter 8 establishes the limits of context free languages via another version of the pumping lemma, and also discusses some closure properties of context free languages.

Context free languages are more expressive than regular languages. In particular, sentences in context free languages can be assigned interesting structures that can be used to convey meaning. In the case of programming languages, these meanings can be seen as chunks of machine code associated with program statements. The disadvantage is that determining whether a string belongs to a context free language, and what structure should be assigned to it (parsing) is intrinsically less efficient than the corresponding task for regular languages.

Chapter 5

Context Free Grammars

5.1 CF Grammars

Recall the general definition of a grammar that we gave earlier:

A grammar is a tuple

$G = \langle N, V, P, S \rangle$

where
N is a set of non-terminal symbols
V is a set of terminal symbols (the vocabulary)
P is a set of productions or rules
$S \in N$ is a priviliged non-terminal start symbol

Rules take the form

LHS $\rightarrow$ RHS

where
LHS is a sequence of terminals and/or non-terminals

RHS is a (possibly empty) sequence of terminals and/or non-terminals

A context free grammar is a grammar with the following additional restriction on the form of the rules

Context Free Grammar:
A context free grammar is a grammar where all rules, LHS $\rightarrow$ RHS, are such that

- The LHS is a single non-terminal symbol
- The RHS is a (possibly empty) sequence of terminal and/or non-terminal symbols

Note that the restrictions on the form of the rules is somewhat laxer than for regular grammars (and that any regular grammar also satisfies the restrictions for a context free grammar).

The terminal symbols constitute the vocabulary of the language. Non-terminal symbols do not occur in sentences of the language defined by the grammar. Instead, their role is to be expanded out into sequences of terminal symbols by the rules of the grammar.

5.1.1 Example CF Grammars

Grammar for $a^n b^n$

Recall that in chapter 4, we showed that the language $a^n b^n$ was not regular. However, it is context free, and here is the grammar for the language:

1. $S \rightarrow aSb$
2. $S \rightarrow ab$

The sentence *aaabbb* can be derived using these two rules as follows.

$\underline{S}$	
$a\underline{S}b$	rule 1
$aa\underline{S}bb$	rule 1
$aaabbb$	rule 2

Recall (p. 8) that a derivation proceeds by first writing down the start symbol of the grammar (in this case, the non-terminal S). Then a rule is selected whose left hand side (LHS) matches the non-terminal symbol. This symbol is replaced by the right hand side (RHS) of the rule. Then another rule is selected to expand out a non-terminal in the new string, shown underlined. This continues until the string contains no more non-terminals to be expanded.

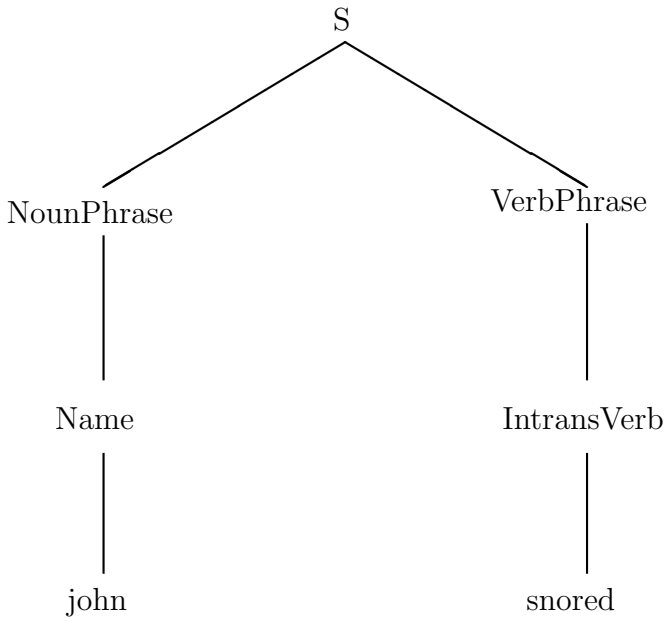
An English-like Grammar

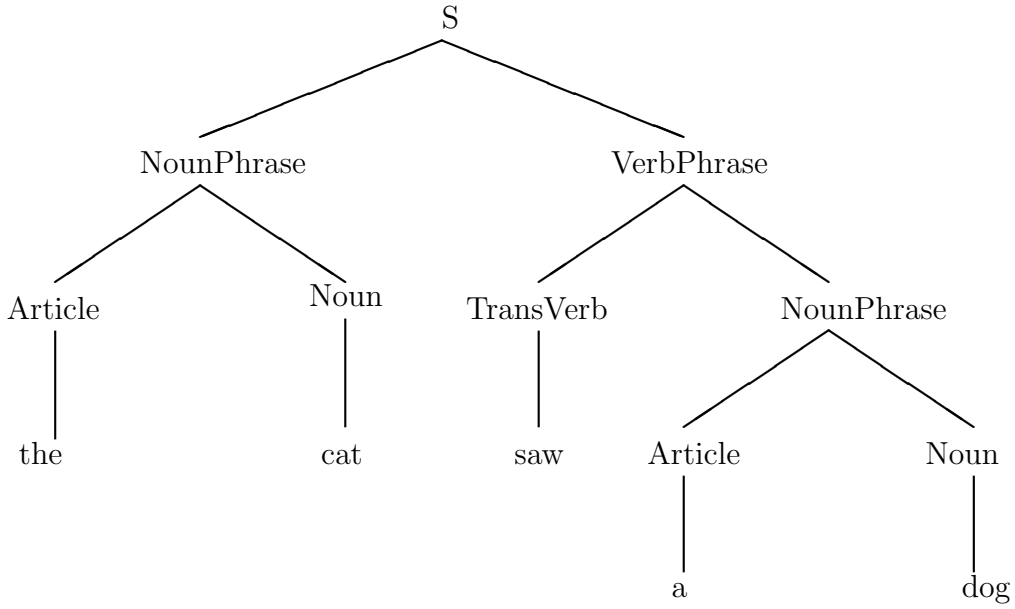
Here is another example of a context free grammar, where S is the start symbol, the non-terminals begin with capital letters and the rules are as follows:

S → NounPhrase VerbPhrase
NounPhrase → Name
NounPhrase → Article Noun
VerbPhrase → IntransVerb
VerbPhrase → TransVerb NounPhrase

Name → john
Name → mary
Article → a
Article → the
Noun → cat
Noun → dog
IntransVerb → slept
IntransVerb → snored
TransVerb → likes
TransVerb → saw

This grammar gives rise to sentences such as the following, which are shown along with derivation trees indicating how the rules of the grammar can be used to construct the sentences.





We will have more to say about derivations and trees in the next section.

Empty Productions

Context free grammars also allow *empty productions*. These are rules where the right hand side is the empty string, ϵ. An alternative grammar for the language $a^n b^n$ involving an empty production is as follows:

1. $S \rightarrow aSb$
2. $S \rightarrow \epsilon$

The sentence *aaabbb* can be derived using these two rules as follows.

$\underline{S}$	
$a\underline{S}b$	rule 1
$aa\underline{S}bb$	rule 1
$aaa\underline{S}bbb$	rule 1
$aaabbb$	rule 2

In the last step of the derivation, the non-terminal S is replaced by the empty string.

5.2 Derivations and Trees

We have informally introduced the ideas of a derivation, and of a tree representing a derivation. Now we will be a little more precise about this. The notion of a derivation is the fundamental concept; trees are a convenient way of representing them.

5.2.1 Derivability

In order to define a derivation, we need first to define *derivability*. Suppose that we have some string of terminal and non-terminal symbols, e.g.

$$uLv$$

where u and v are sequences of terminal and non-terminal symbols, and L is some sequence of symbols that matches the left hand side of a grammar rule, $L \rightarrow R$. We can replace the L by the R to derive a new sequence uRv. We write this derivation as

$$uLv \Rightarrow uRv$$

It may well be that the string uRv contains another substring matching the left hand sides of another rule, so that some further string, w is derivable from uRv. The string, w, is thus derivable from uLv via the application of zero one or more rules, and we write this as

$$uLv \stackrel{*}{\Rightarrow} w$$

Turning to the formal definition of the derivability relations, $\stackrel{*}{\Rightarrow}$, this is given recursively as follows

Definition of Derivability, $\stackrel{*}{\Rightarrow}$

Base Case $v \stackrel{*}{\Rightarrow} v$, for any string $v \in (V \cup N)^*$

Recursion If $u = xLy$, $v \stackrel{*}{\Rightarrow} u$, and $L \rightarrow R$ is a grammar rule, then $v \stackrel{*}{\Rightarrow} xRy$

Note: the notation $u \Rightarrow v$ means that v is derivable from u through the application of just one grammar rule. $u \stackrel{*}{\Rightarrow} v$ means that v is derivable from u through the application of zero, one or more grammar rules. The notation $u \stackrel{+}{\Rightarrow} v$ means that v is derivable from u through the application of one or more grammar rules.

5.2.2 Sentential Forms

A *sentential form* is any string of terminal and/or non-terminal symbols that can be derived from the start symbol of the grammar:

Sentential Form:
If $S \stackrel{*}{\Rightarrow} u$, where S is the start symbol of the grammar, then
u ($u \in (V \cup N)^*$) is a sentential form of the grammar

A sentence of a grammar is a sentential form consisting only of terminal symbols.

5.2.3 Left- and Rightmost Derivations

Derivations

A derivation is sequence of strings of terminal and/or non-terminal symbols, where each string is derivable from the preceding one by means of one grammar rule. The first string in the derivation comprises just the start symbol of the grammar, and the last string must be a sentence of the grammar. Thus

Derivation
A derivation is a sequence of strings in $(V \cup N)^*$:

$$S, u_1, \ldots, u_i, \ldots u_n$$

where
(a) for every i, $u_{i-1} \Rightarrow u_i$, and
(b) $u_n \in V^*$

We have been writing down derivations in the following way

$\underline{S}$	
$a\underline{S}b$	rule 1
$aa\underline{S}bb$	rule 1
$aaa\underline{S}bbb$	rule 1
$aaabbb$	rule 2

Where each line consists of s string of terminals and/or non-terminals derivable from the last, beginning with the start symbol and ending with a string of terminal. We have also been indicating the rule used at each step of the derivation, and to which (underlined) part of the string the rule applies.

[Exercise: show that every string in a derivation is a sentential form of the grammar]

Leftmost Derivations

For a context free grammar, each step in the derivation will expand out just one non-terminal symbol in the string. (This is because only single non-terminals may stand on the left hand sides of grammar rules). Sometimes, a string at an intermediate point in a derivation will contain more than one non-terminal, meaning that there is a choice about which one to expand.

For example, consider the grammar

1. $S \rightarrow AB$
2. $A \rightarrow a$
3. $B \rightarrow CD$
4. $C \rightarrow c$
5. $D \rightarrow d$

The first step in any derivation will be

$\underline{S} \Rightarrow AB$ (rule 1)

At this point, we could either expand the A (the leftmost non-terminal) or the B (the rightmost non-terminal).

Here is a derivation in which we consistently expand the leftmost non-terminal symbol:

	S	$\Rightarrow AB$	rule 1
		$\Rightarrow aB$	rule 2
Leftmost derivation:		$\Rightarrow aCD$	rule 3
		$\Rightarrow acD$	rule 4
		$\Rightarrow acd$	rule 5

And here is a derivation where we consistently expand the rightmost non-terminal:

$$
\begin{array}{lll}
 & S \Rightarrow AB & \text{rule 1} \\
 & \Rightarrow ACD & \text{rule 3} \\
\text{Rightmost derivation:} & \Rightarrow ACd & \text{rule 5} \\
 & \Rightarrow Acd & \text{rule 4} \\
 & \Rightarrow acd & \text{rule 2}
\end{array}
$$

There is a sense in which these two derivations are the same: the same rules have been used to expand out the same non-terminals to lead to the same final terminal string. The only difference is in the relative order in which the non-terminals have been expanded.

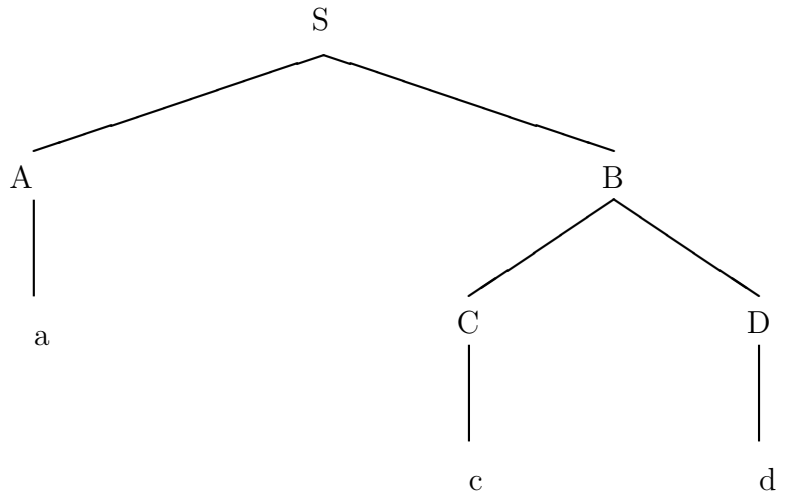
This observation can be generalized:

> For any derivation of a string in a context free grammar, there exists a leftmost derivation of that string.

We will not prove this result here. What it means is that leftmost derivations can be seen as the canonical form for doing derivations in context free grammars.

5.2.4 Derivation Trees

Derivation trees are another way of representing derivations, and abstract away from alternative orders in which rules can be applied. For example, the following tree represents both the leftmost and rightmost derivations of the string *acd* shown previously



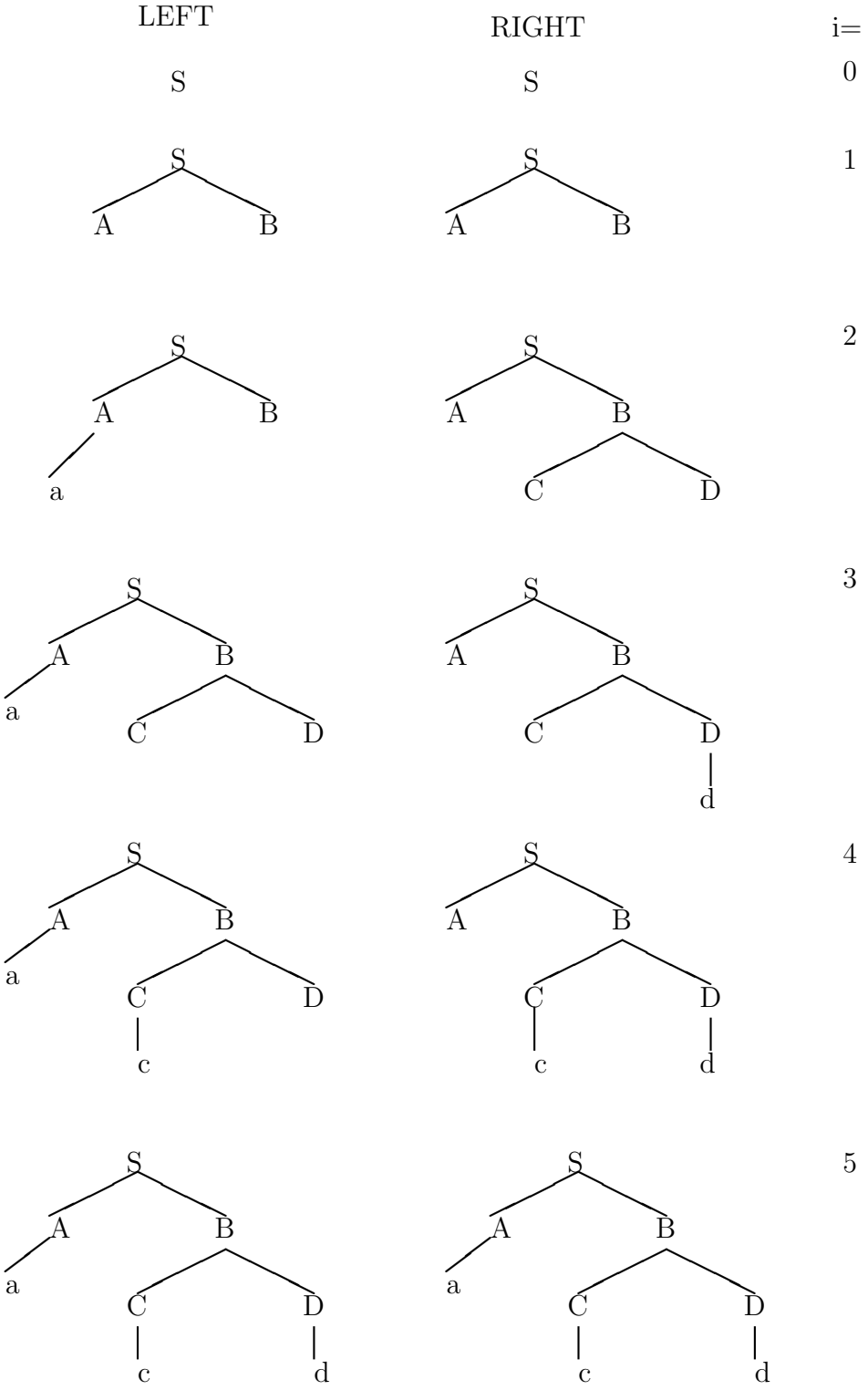
Given some derivation in a grammar, $S \stackrel{*}{\Rightarrow} w$, we can construct a derivation tree, DT, for it as follows:

1. Initialize DT with the root node S
2. For $i = 1$ to the length of the derivation, do:

> If $A \to x_1 x_2 \ldots x_n$ is the rule used in the ith step of the derivation,
> and is applied to the sentential form uAv,
> add $x_1, x_2, \ldots, x_n$ as children of A in the tree

Derivation trees are a way of grouping derivations together into equivalence classes, where the only difference between the derivations are the orders in which the rules are applied.

Below we show how both the rightmost and leftmost derivations for the string *acd* give rise to the same derivation tree:


LEFT
RIGHT
i=
S
0
S
S
1
A
B
A
B
S
2
A
B
a
S
A
B
C
D
S
3
A
B
a
C
D
S
A
B
C
D
d
S
4
A
B
a
C
D
c
S
A
B
C
D
c
d
S
5
A
B
a
C
D
c
d
S
A
B
a
C
D
c
d

5.3 Ambiguity

A sentence is ambiguous if there is more than one leftmost derivation of the sentence in the grammar. Or put another way, it is ambiguous if there is more than one derivation tree. In English, ambiguous sentences typically convey more than one meaning, e.g.

> Women like chocolate more than men
> — Either: Women like chocolate more than men like chocolate
> — Or: Women like chocolate more than they like men

However, ambiguity is not defined in terms of conveying more than one meaning. It is defined solely in terms of a sentence having more than one grammatical derivation:

A sentence is *ambiguous* in a grammar G iff there is more than one derivation tree (or leftmost derivation) for the sentence in G

As an example of ambiguity, consider the following grammar: Consider the ambiguous grammar

$S \rightarrow AB$
$A \rightarrow a$
$A \rightarrow ac\ B \rightarrow cd$
$B \rightarrow d$

There are two distinct derivation trees for $S \stackrel{*}{\Rightarrow} acd$

This means that the sentence acd is ambiguous in the grammar just given. However, the same sentence is unambiguous in our earlier grammar:

$S \rightarrow AB$
$A \rightarrow a$
$B \rightarrow CD$
$C \rightarrow c$
$D \rightarrow d$

where there is only one derivation. Thus, the ambiguity of a sentence is alway relative to a particular grammar.

Bearing this in mind, we can classify three levels of ambiguity

1. Sentence:
   A sentence is ambiguous in a grammar G iff there is more than one derivation of the sentence in g

2. Grammar:
   A grammar is ambiguous iff there is some sentence in the language it defines that is ambiguous

3. Language:
   A language is ambiguous iff it is not possible to give an unambiguous grammar for the language.

In connection with unambiguous languages, we should note that the language comprising the single string acd is unambiguous, even though we managed to concoct an ambiguous grammar for the language. This is because we can also give an unambiguous grammar for the language.

As another example of an unambiguous language with at least one ambiguous grammar, consider the following two grammars for the language a^+

Ambiguous	Unambiguous
$S \rightarrow aS$	$S \rightarrow aS$
$S \rightarrow Sa$	
$S \rightarrow a$	$S \rightarrow a$

[Exercise: show that the first grammar is ambiguous]

There are a number of languages that are inherently ambiguous: English is one of them. But programming languages are deliberately designed to be unambiguous.

5.4 Recursion in Grammars

Recursive Rule:
A recursive rule takes the form

$$A \rightarrow uAv$$

(where u and v are any sequence of terminals and non-terminal symbols, including the empty sequence

A recursive rule in a grammar is one where expanding out the non-terminal on the left hand side of the rule reintroduces another copy of the same symbol on the right hand side of the rule.

Just as any recursive definition or program procedure requires as a base case, so recursive rules in grammars also typically require non-recursive rules. Given a recursive rule like $A \rightarrow uAv$, there ought also to be a non-recursive rule expanding A, e.g. $A \rightarrow ab$. (The absence of base rules for recursive rules does

not stop something from being a grammar, but it does render the recursive rules both pointless and harmful).

Recursion is the means by which a finite grammar can generate a language containing an infinite number of strings. Recall the grammar for the language $a^n b^n$:

1. $S \rightarrow aSb$
2. $S \rightarrow ab$

This consists of one recursive rule for S (rule 1) and one non-recursive rule (rule 2). We can use repeated applications of rule 1 to generate strings of as and bs as long as we like:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow \ldots \Rightarrow aaaa \ldots S \ldots bbbb$$

This is terminated by using rule 2 to remove the non-terminal S from the string:

$$S \stackrel{*}{\Rightarrow} aaaa \ldots S \ldots bbbb \Rightarrow aaaa \ldots ab \ldots bbbb$$

5.4.1 Left- and Right-Recursion

A *left-recursive rule* is one where the symbol on the left hand side is the first to appear on the right hand side. (Left recursive rules can be problematic for top-down parsing algorithms — Chapter 7). *Right-recursive rules* are ones where the symbol on the left hand side is the last one to appear on the right hand side. Thus

- Left recursive rule: $A \rightarrow Au$
- Right recursive rule: $A \rightarrow uA$

5.4.2 Indirect Recursion

Sometimes the recursion in a grammar can be spread across several rules. Consider the grammar for the language $a^{2n} b^{2n}$

1. $S \rightarrow aTb$
2. $T \rightarrow aSb$
3. $T \rightarrow ab$

None of the rules are individually recursive, but if we trace a derivation for $aaaabbbb$ through, we get

$$S \Rightarrow aTb \Rightarrow aaSbb \Rightarrow aaaTbbb \Rightarrow aaaabbbb$$

In expanding out the symbol S, we do not immediately reintroduce a copy of S; but at some later stage in the derivation a copy of S is reintroduced. Similarly for T. This is an example of an indirectly recursive derivation

Indirectly recursive derivation:
A derivation $A \Rightarrow w \stackrel{+}{\Rightarrow} uAv$, where A is not in w, is indirectly recursive.

It is also helpful to talk about which non-terminal symbols are recursive, as well as which rules are recursive:

Recursive Symbols:
A non-terminal, A, is recursive, if there is some derivation $A \stackrel{+}{\Rightarrow} uAv$

A symbol A can be recursive, even if there is no directly recursive rule expanding A.

5.5 Normal Forms for CFGs

In this section we look at three alternative ways of representing context free grammars:

- Backus-Naur Form (BNF)
- Chomsky Normal Form (CNF)
- Greibach Normal Form (GNF)

Backus-Naur form is merely an alternative notation for writing context free grammars. Chomsky and Greibach normal forms, by contrast, impose additional restrictions on the form that the grammar rules must take. However, any arbitrary context free grammar can be converted to either CNF or GNF. The converted normal form grammars will define exactly the same language as the original grammar, but may assign different derivation trees to the sentences in the language.

5.5.1 Backus-Naur Form

Backus-Naur Form (BNF) is *not* a normal form. It is just an alternative way of writing down context free grammars. BNF was devised for the specification of the programming language ALGOL, and has been widely used in programming language specifications since.

The main notational changes are:

- Instead of writing $\rightarrow$, in BNF you write ::=
- Non-terminal symbols are enclosed in angle brackets, $\langle\rangle$
- Different expansions for a given non-terminal are written in one rule, with the alternatives separated by |

To see what these changes mean, consider the following grammar in our original notation, and the same grammar in BNF

Original Notation

$S \to AB$
$S \to Ac$
$S \to A$
$A \to a$
$A \to z$
$B \to Bb$
$B \to b$

Equivalent in BNF

$\langle S \rangle ::= \langle A \rangle \mid \langle A \rangle \langle B \rangle \mid \langle A \rangle c$
$\langle A \rangle ::= a \mid z$
$\langle B \rangle ::= b \mid \langle B \rangle b$

Extended BNF

Extended BNF (EBNF) adds another constructs to BNF

- Strings enclosed in braces, {} indicate zero or more repetitions

In EBNF, we can express the previous CFG grammar as

$\langle S \rangle ::= \langle A \rangle \{b\} \mid \langle A \rangle c$
$\langle A \rangle ::= a \mid z$

Note that conversion from CFG to EBNF is somewhat more complex than from CFG to BNF, but it can always be done.

5.5.2 Chomsky Normal Form

A grammar is in Chomsky Normal Form (CNF) if its rules have the following forms

Chomsky Normal Form:
All rules must take one of the following forms:

1. $X \to YZ$
   Where X, Y and Z are all non-terminals, and neither Y nor Z are the start symbol
2. $X \to a$
   Where X is a non-terminal and a is a terminal
3. $S \to \epsilon$
   Where S is the start symbol
   (there will be at most one rule of this form, and only if $\epsilon \in L(G)$).

A grammar is in Chomsky Normal Form if every non-terminal expands to either a pair of non-terminals, or a single terminal. This means that derivation trees for grammars in CNF will always be binary trees. Moreover, at most one empty production, $S \to \epsilon$ is allowed.

We will shortly establish that any CF grammar can be converted to a CNF grammar defining exactly the same language. Chomsky Normal Form is useful for establishing certain properties of grammars (see the chapter on the pumping lemma).

Conversion to CNF

Converting a grammar to Chomsky Normal Form proceeds in a certain number of steps:

1. Make the start symbol non-recursive
   i.e. S does not occur in the RHS of any rules
2. Remove empty productions: $X \to \epsilon$
3. Remove chain rules: $X \to Y$
4. Remove useless symbols: useless symbols are ones that do not occur in any successful derivations of sentences
5. Remove secondary productions: $X \to RHS$ where RHS is a mixture of terminal and non-terminal symbols
6. Remove tertiary productions: $X \to Y_1 Y_2 \dots Y_m$, where $m > 2$ and $Y_1, Y_2, \dots, Y_m$ are non-terminals

These steps should be followed in this order. We now describe each step in turn. Note that each step preserves the language generated by the transformed grammar.

[Also note: you may expect exam questions asking you to convert context free grammars to Chomsky Normal Form.]

1. Make Start Symbol Non-Recursive

To make the start symbol, S, non-recursive we have to make sure that it does not occur on the right hand side of any rule in the grammar. It may be that this is already the case with the grammar.

However, in the grammar

1. $S \rightarrow aTb$
2. $T \rightarrow aSb$
3. $T \rightarrow ab$

the start symbol S is not recursive. We therefore add a new start symbol, S' and a rule $S' \rightarrow S$, so that the grammar is now

1. $S' \rightarrow S$
2. $S \rightarrow aTb$
3. $T \rightarrow aSb$
4. $T \rightarrow ab$

The new start symbol does not occur on the right hand side of any rules, and the new grammar generates the same strings as the old one.

To summarise

To make the start symbol non-recursive:

(a) Create a new start symbol, S'

(b) Add the rule $S' \rightarrow S$ to the grammar

2. Remove Empty Productions

The next step involves removng all empty productions from the grammar, with the possible execption of $S \rightarrow \epsilon$, where S is the start symbol. To do this, we first have to determine the *nullable* symbols of the grammar — the non-terminal symbols from which the empty string can be derived.

Determining the nullable symbols:
The non-terminal A is nullable iff $A \stackrel{*}{\Rightarrow} \epsilon$
Procedure for building set of nullable symbols, NS

1. Init: Let $NS = \{A \mid A \rightarrow \epsilon \text{ is a grammar rule}\}$
2. Repeat until no new symbols are added to NS:
   For each rule $B \rightarrow \alpha$, if all the symbols in α are in NS, add B to NS (if it not already there)

For example, consider the grammar

$$S \to ACA$$
$$A \to aAa \mid B \mid C$$
$$B \to bB \mid b$$
$$C \to cC \mid \epsilon$$

The set of nullable symbols is initialised to $\{C\}$. The rule $A \to C$ means that A is also nullable, and so gets added to the set. This in turns means that the rule $S \to ACA$ shows S to be a nullable symbol.

Having determined which non-terminal symbols are nullable, we now have to modify the rules of the grammar. Each rule in the grammar

$$B \to X_1 X_2 \ldots X_n$$

has to be replaced by a set of rules obtained by striking out zero or more of the nullable symbols from the right hand side, $X_1 X_2 \ldots X_n$. However, if all of $X_1 X_2 \ldots X_n$ are nullable, and B is not the start symbol, we do *not* strike out all of the right hand side to give a rule $B \to \epsilon$

Example Suppose B is nullable, and we have a production $A \to BCBd$

We strike out the nullable symbols to get the following rules

$$A \to CBd$$
$$A \to BCd$$
$$A \to Cd$$
$$A \to BCBd$$

These correspond to instances of applying the original rule in derivations where

a) the first occurrence of B derives ϵ
b) the second occurrence of B derives ϵ
c) the first and second occurrences of B derive ϵ
d) neither occurrence of B derives ϵ

It should be apparent that the four new rules can be used to derive exactly the same strings as the original rule: no more and no less.

Moreover, the rules will still generate the same strings, even if there are no ϵ-rules in the grammar. For in the cases where an ϵ-rule was used in deriving ϵ from an occurrence of B in the original rule, the additional rule just missing that occurrence out can be used to derive the same string

3. Remove Chain Rules

A chain rule takes the form

$$A \to B$$

where both A and B are non-terminals. Given such a rule we may as well skip out the B, and rewrite A to whatever B rewrites to.

For example, if we have

$$A \to B$$
$$B \to CD$$
$$B \to E$$

why not replace $A \to B$ with $A \to CD$ and $A \to E$? This introduces a second chain rule, $A \to E$, so we would also want to replace this.

A derivation $A \stackrel{*}{\Rightarrow} C$ consisting solely of chain rules is called a chain. We need an algorithm to determine all possible non-terminals occurring in chains starting from A, call this $CHAIN(A)$:

Terminals derivable by chains from A:

1. Init:
   $CHAIN(A) = \{A\}$, $PREV = \{\}$
2. Repeat until $CHAIN(A) = PREV$:
   (a) $NEW = CHAIN(A) - PREV$
   (b) $PREV = CHAIN(A)$
   (c) For each non-terminal, $B \in NEW$, do
   If there is a rule $B \to C$, then
   $CHAIN(A) = CHAIN(A) \cup \{C\}$

In other words, just follow all possible chains down from A, collecting the non-terminals C that occur in them

Having collected $CHAIN(A)$ for every non-terminal A, we update the grammar rules as follows:

Removing Chain Rules:

1. Remove all chain rules, $A \to B$ from the grammar
2. If $B \in CHAIN(A)$ and there is a non-chain rule $B \to \alpha$
   Add the rule $A \to \alpha$ to the grammar

Again it should be possible to see that the new grammar generates exactly the same set of strings as the old one.

Example Suppose we have the grammar

$$S \to CA \mid A \mid C$$
$$A \to aAa \mid B \mid C$$
$$B \to bB \mid b$$
$$C \to cC \mid c$$

Then we calculate the following chain sets:

$$CHAIN(S) = \{S, A, C, B\}$$
$$CHAIN(A) = \{A, B, C\}$$
$$CHAIN(B) = \{B\}$$
$$CHAIN(C) = \{C\}$$

These sets are used to generate the rules:

$$S \to CA \mid aAa \mid bB \mid b \mid cC \mid c$$
$$A \to aAa \mid bB \mid b \mid cC \mid c$$
$$B \to bB \mid b$$
$$C \to cC \mid c$$

4. Remove Useless Symbols

A symbol is useful if it takes part in the generation of some sentence in the language.

A symbol $x \in (V \cup N)$ is *useful* if there is a derivation

$$S \stackrel{*}{\Rightarrow} uxv \stackrel{*}{\Rightarrow} w$$
where $u, v \in (V \cup N)^*$ and $w \in V^*$

Otherwise, the symbol is *useless*

Two conditions must be satisfied for a non-terminal to be useful

1. It must occur in a sentential form of the grammar
2. There must be a derivation of a terminal string from the symbol

Rules containing useless symbols can never take part in the derivation of a sentence, so we want to eliminate them from the grammar.

There are a number of reasons why useless symbols may occur in a grammar

- Sloppiness in writing the grammar or modifying a large grammar over the course of time, so that it includes redundant rules
- Pathological cases of eliminating ϵ-rules where there are some non-terminals that only derive ϵ. In a grammar without ϵ-rules, such non-terminals will be useless.

Detecting useless symbols divides into two parts

1. Detecting which symbols can(not) derive terminal strings
2. Detecting which symbols (don't) occur in sentential forms

Algorithm for collecting non-terminals deriving terminal strings, $TERM$

1. Init: $TERM = \{A \mid \text{there is a rule } A \to w, \text{ where } w \in V^*\}$
2. repeat:
   (a) $PREV = TERM$
   (b) For each non-terminal A, do
   If there is a rule $A \to w$ and $w \in (PREV \cup V)*$
   then $TERM = TERM \cup \{A\}$

   until $PREV = TERM$

In other words, start by collecting those non-terminals that expand directly to terminal strings. Then, iteratively, if there is a non-terminal that expands to a string of terminals plus symbols that in turn expand to terminal strings, add it to the list $TERM$

The first stage in removing useless symbols from a grammar is to identify all those non-terminals not in $TERM$, i.e. non-terminal symbols from which no terminal strings can be derived. Any rule containing one of more of these useless symbols should be removed from the grammar.

Having removed all the useless non-terminals from which no terminal string can be derived, we now determine which non-terminals occur within sentential forms of the (revised) grammar.

Algorithm for collecting non-terminals occurring in sentential forms, $SFNT$

1. Init: $SFNT = \{S\}$, $PREV = \{\}$
2. Repeat until $SFNT = PREV$
   (a) $NEW = SFNT - PREV$
   (b) $PREV = SFNT$
   (c) For each $A \in NEW$ do
   For each rule $A \to w$ do
   add all non-terminals in w to $SFNT$

In other words, start by getting sentential forms from one rule application to S. For each non-terminal A, added to $SFNT$ by this, look at all possible rules expanding A, and add any new non-terminals to $SFNT$. $PREV$ is used to ensure that in each iteration, we only expand the non-terminals added to

$SFNT$ on the last step. That is, we don't repeatedly expand all the symbols in $SFNT$, only the new ones.

Any non-terminal not occurring in $SFNT$ is useless. Any rule containing such a useless symbol should be removed from the grammar.

Note: it is important that we first remove rules containing non-terminals that do not generate terminal strings, and only then remove rule containing non-terminals not occurring in sentential forms.

Example Consider the grammar

$$S \to a \mid AB \mid aC$$
$$A \to b$$
$$C \to cC$$

First we look for non-terminals that cannot generate terminal strings. Clearly B is such a useless symbol, since there is no rule expanding it. Less clearly, C is also useless: the only rule expanding it is a recursive one and there is no further non-recursive rule for C.

Hence, we first change the grammar to

$$S \to a$$
$$A \to b$$

Here, it is clear that A does not occur in any sentential form of the grammar, so we remove any rules containing A to give

$$S \to a$$

Question: what would we get if we reversed the two stages, and first removed those symbols not occurring in sentential forms?

5. Remove Secondary Productions

Secondary productions are rules of the form

$$X \to RHS$$

where RHS is a mixture of terminal and non-terminal symbols For example, a rule like

$$A \to bDcF$$

To remove secondary productions, replace the terminal symbols by new non-terminal symbols (B' and C') And add rules rewriting the new non-terminals to the terminals they replace.

Thus the rule above is expanded to:

$A \to B'DC'F$
$B' \to b$
$C' \to c$

Again, it should be clear that the new grammar generates the same language as before.

6. Remove Tertiary Productions

Tertiary productions are rules of the form

$X \to Y_1 Y_2 \dots Y_m$
where $m > 2$ and all Y_is are non-terminals

To remove tertiary productions, we again introduce new non-terminal symbols, and rules rewriting them to successive suffixes of the RHS.

For example

$A \to B'DC'F$

is replaced with

$A \to B'X'$
$X' \to DY'$
$Y' \to C'F$

Again, it should be clear that the new grammar generates the same language as before

Full Example

We now give an example of a full conversion of a context free grammar to Chomsky Normal Form. Consider the grammar

$S \to SaB \mid aB \mid BC$
$B \to bB \mid \epsilon$
$C \to BC \mid AD$
$A \to bC \mid bA$
$D \to d$

The conversion proceeds as follows

1. Add new non-recursive start symbols, S':

   $S' \to S$
   $S \to SaB \mid aB \mid BC$

$B \to bB \mid \epsilon$
$C \to BC \mid AD$
$A \to bC \mid bA$
$D \to d$

2. B is the only nullable symbol. Make extra copies of rules containing B with the B struck out:

$S' \to S$
$S \to SaB \mid Sa \mid aB \mid a \mid BC \mid C$
$B \to bB \mid b$
$C \to BC \mid C \mid AD$
$A \to bC \mid bA$
$D \to d$

3. Remove chain rules ($S' \to S$, $S \to C$, $C \to C$):

$S' \to SaB \mid Sa \mid aB \mid a \mid BC \mid AD$
$S \to SaB \mid Sa \mid aB \mid a \mid BC \mid AD$
$B \to bB \mid b$
$C \to BC \mid AD$
$A \to bC \mid bA$
$D \to d$

4. Remove useless symbols

(a) The symbols A and C cannot derive terminal strings: their rules form a mutually unterminated recursion. Hence remove all rules containing A or C:

$S' \to SaB \mid Sa \mid aB \mid a$
$S \to SaB \mid Sa \mid aB \mid a$
$B \to bB \mid b$
$D \to d$

(b) The symbols D and d are now not derivable from S', so these are useless and must be removed:

$S' \to SaB \mid Sa \mid aB \mid a$
$S \to SaB \mid Sa \mid aB \mid a$
$B \to bB \mid b$

5. Remove secondary productions:

$S' \to SA'B \mid SA' \mid A'B \mid a$
$S \to SA'B \mid SA' \mid A'B \mid a$
$B \to B'B \mid b$
$A' \to a$
$B' \to b$

6. Remove tertiary productions:

$S' \to SX \mid SA' \mid A'B \mid a$
$S \to SX \mid SA' \mid A'B \mid a$
$B \to B'B \mid b$
$A' \to a$
$B' \to b$
$X \to A'B$

This grammar is now in CNF

5.5.3 Greibach Normal Form

A grammar is in Greibach Normal Form if it rules have the following forms

Greibach Normal Form
Every rule is in one of the following forms

1. $X \to aY_1 \dots Y_n$
   Where a is a terminal and $X, Y_1, \dots, Y_n$ are non-terminals, and none of $Y_1, \dots, Y_n$ are the start symbol
2. $S \to \epsilon$
   Where S is the start symbol
   (there will be at most one rule of this form, and only if $\epsilon \in L(G)$).

For every context free grammar, there is an equivalent grammar in GNF defining the same language.

Greibach Normal Form eliminates all left recursion from a grammar. (A left recursive rule is of the form $A \to Au$) Left recursion can cause problems in top-down parsing, as we will see later, and so eliminating it can be helpful. GNF is also useful when it comes to showing how any context free grammar has an equivalent pushdown automaton (chapter 6).

We will not concentrate on the full range of transformations that take you from a grammar in CNF to one in GNF (for details see Sudkamp 5.5 & 5.6; Hopcroft & Ullman chap 4.6; or Rayward-Smith chap 5). We will focus on just one of the steps: removing left-recursion from a grammar

Removing Left-Recursion

A left-recursive rule has the form $A \to A\alpha$

To eliminate left-recursive rules

1. Let $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_r$ be all the left-recursive rules for A.
2. Let $A \rightarrow \beta_1 \mid \beta_2 \mid \ldots \beta_s$ be the remaining A-rules
3. Add a new non-terminal symbol, B
4. Remove all the A-rules (left-recursive and others)
5. Replace them by
   (a) For $1 \leq i \leq s$
   $A \rightarrow \beta_i$
   $A \rightarrow \beta_i B$
   (b) For $1 \leq i \leq r$
   $B \rightarrow \alpha_i$
   $B \rightarrow \alpha_i B$

Example Consider the grammar:

$A \rightarrow Aa$
$A \rightarrow c$

This becomes

$A \rightarrow c$
$A \rightarrow cB$

$B \rightarrow aB$
$B \rightarrow a$

Basically, this replaces left-recursion with right-recursion Consider a left-recursive derivation in the original grammar:

$A \Rightarrow Aa \Rightarrow Aaa \Rightarrow caa$

Compare this with the right-recursive derivation in the new grammar:

$A \Rightarrow cB \Rightarrow caB \Rightarrow caa$

Example of GNF Conversion

Let us take the grammar that we previously converted to CNF, at the point just after eliminating useless symbols. That is:

$S' \rightarrow SaB \mid Sa \mid aB \mid a$
$S \rightarrow SaB \mid Sa \mid aB \mid a$
$B \rightarrow bB \mid b$

We eliminate the left recursive rules $S \to SaB \mid Sa$, introducing a new right recursive symbols Z as follows:

$$
\begin{aligned}
&S' \to SaB \mid Sa \mid aB \mid a \\
&B \to bB \mid b \\
&S \to aBZ \mid aZ \mid aB \mid a \\
&Z \to aBZ \mid aZ \mid aB \mid a
\end{aligned}
$$

(In this particular case, the rules for expanding Z are identical for the rules for expanding S. We could therefore safely replace all occurrences of Z by S, and omit the Z rules:

$$
\begin{aligned}
&S' \to SaB \mid Sa \mid aB \mid a \\
&B \to bB \mid b \\
&S \to aBS \mid aS \mid aB \mid a
\end{aligned}
$$

However, in general eliminating left recursion will not lead to an eliminable right recursive symbol)

We can remove non-initial terminal symbols from the right hand sides of rules in the same was as for removing secondary productions:

$$
\begin{aligned}
&S' \to SA'B \mid SA' \mid aB \mid a \\
&B \to bB \mid b \\
&S \to aBS \mid aS \mid aB \mid a \\
&A' \to a
\end{aligned}
$$

At this point, all the rules except $S' \to SA'B \mid SA'$ are in GNF (i.e. right hand side begins with a terminal followed by zero or more non-terminals). We can plug in the expansions of S to correct this:

$$
\begin{aligned}
&S' \to aBSA'B \mid aSA'B \mid aBA'B \mid aA'B \\
&\quad \mid aBSA' \mid aSA' \mid aBA' \mid aA' \\
&\quad \mid aB \mid a \\
&S \to aBS \mid aS \mid aB \mid a \\
&A' \to a
\end{aligned}
$$

The grammar is now in GNF.

5.6 Summary

This chapter introduced context free languages, as defined in terms of context free grammars (CFGs). Context free languages are one step above regular languages in the Chomsky hierarchy. They are of considerable practical importance; for example, the syntax of programming languages is context free. Natural languages like English are also (nearly) context free.

We first described the restrictions on rules that make them rules of a context-free grammar. We then introduced the notions of derivations, derivation trees and ambiguity. Finally we discussed Chomsky and Greibach normal forms for CFGs, and how to convert any CFG into these normal forms. The conversion preserves the language generated by the grammar, but may alter the derivations associated with individual strings within the language.

Chapter 6

Push-Down Stack Automata

Context free language and grammars are associated with a particular kind of automaton, just as regular languages and grammars are associated with finite state machines. In the case of context free languages, the corresponding automaton is a (non-deterministic) *push-down stack automaton*. This is essentially a finite state automaton with an additional stack-like memory.

In this chapter, section 6.1 introduces push-down stack automata (PDAs), and comments on the non-equivalence between deterministic and non-deterministic PDAs. Section 6.2 shows how any context free grammar can be converted to a (non-deterministic) PDA accepting the same language, and section 6.3 describes the conversion in the opposite direction.

6.1 Push-Down Stack Automata

A push-down stack automaton is essentially a finite state machine with an additional stack like memory. Transitions are dependent on

1. the current state
2. the next word of the input
3. the item at the top of the stack

Taking a transition will

1. Move you to a possibly different state
2. Consume a word of input (unless it is an empty transition)
3. Pop the top item off the stack
   (Note: it is permissible to pop an empty item off the top of the stack, leaving it unchanged)

4. Push a new item onto the top of the stack
   (Note: it is permissible to push an empty item onto the top of the stack, leaving it unchanged)

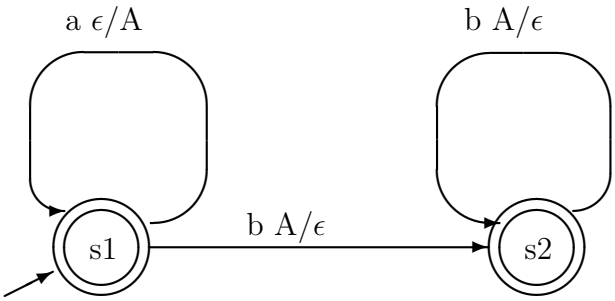
A string is accepted by a PDA if you can take a sequence of transitions, (a) beginning at the start state with an empty stack, (b) ending in an accepting state with an empty stack, and (c) consuming all the words in the string.

6.1.1 Two Examples

PDAs are graphically represented in the same way as finite state automata, but with somewhat more elaborate labels on the arcs (transitions). Each arc is labelled with

(i) the word of input to be consumed
(ii) the item that must be popped off the top of the stack
(iii) the item to be pushed onto the top of the stack when the transition to the next state is made

Here for example is a PDA to accept the language $a^n b^n$:



The transition looping round on state $s1$ is triggered by having the word a as input, and being able to pop the empty item, ϵ off the stack. (Note that you can always pop the empty item off any stack). On consuming the word a and poppng ϵ off the stack, the transition pushes the symbol A onto the top of the stack, and moves back to state $s1$.

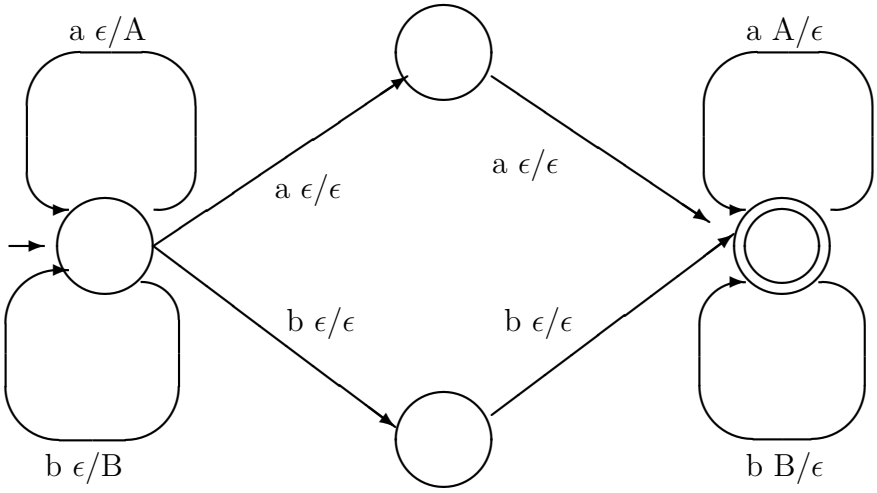
That is, every time an a is encountered in state $s1$, a symbol A is pushed onto the stack, and we return to the state $s1$. Thus after comsuming n occurrences of a in the input string, the stack will contain n instances of the symbol A.

The transition from state $s1$ to $s2$ is triggered by finding b as the next word of input, and having an A on the top of the stack. The transition pops off the topmost A, and does not push anything onto the stack to replace it. Thereafter, transitions loop around state $s2$, consuming bs from the input and popping As off the stack.

The accepting configuration for a PDA is to be in an accepting state ($s2$) with no remaining input and nothing on the stack. Therefore, if there were n

transitions looping round on state $s1$ cnosuming as and pushing As onto the stack, these must be followed by n transitions consuming bs and popping the As off the stack. The stack acts as a memory, keeping track of how many as occurred in the first half of the string.

Here is another PDA, this time to accept even length palindromes over the vocabulary $\{a, b\}$ (a palindrome is a string that reads the same forwards as backwards).



Consider the palindrome *ababaababa*. The first part of the string, *abab* is consumed by looping around on the first state. This will push A, then B then A and then B onto the stack, so that once the input *abab* has been consumed, the stack will contain the symbols $BABA$. We then consume two as, leaving the stack unchanged, and move to the final state. We then loop round, first popping a B off the stack and consuming a b, then an A and an a, then a B and a b, and finally an A and an a. In other words, this produces the last part of the string *baba*, which is the first part in reverse order.

Note that the PDA for $a^n B^n$ is deterministic: at no point is there the possibility for choice about which transition to take next. But the PDA for even length palindromes is non-deterministic: in the initial state with input a once can either pop ϵ off the stack and loop pushing A onto the stack; or one can pop ϵ off the stack, and move to a new state while pushing ϵ onto the stack.

6.1.2 PDAs: Formal Definition

Stacks

First, we will define a stack. For our purposes, a stack is just a string of symbols taken from some stack vocabulary, Γ. For example, if $\Gamma = \{A, B\}$, then possible stack values are

$$AABA$$

BBA
ϵ

where the first symbol in the string counts as the top of the stack, and ϵ is the empty stack.

The two main operations on a stack are (i) to pop a symbol off the top of the stack, returning the symbol and the stack minus the topmost symbol, and (ii) to push a symbol onto the top of the stack. The transitions in PDAs always specify which symbol needs to be popped or pushed. Thus, if tS is a string representing a stack where t is the first symbol and S the remainder or the stack, we have

$$POP(a, tS) = \langle a, S\rangle \text{ if } a = t$$
$$\text{undefined otherwise}$$

$$PUSH(a, S) = aS$$

Note that POP returns a pair of values. Note also that since $S = \epsilon S$, it is always possible to pop the empty string off the top of a stack. Similary, ϵ can be pushed onto a stack without changing the stack.

PDA Definition

A (deterministic) Push-Down Stack Automaton, PDA
is a 7-tuple

$$PDA = \langle S, V, \Gamma, \delta, s, \gamma, F\rangle$$

where

- S is a set of of states
- V is the input vocabulary
- Γ is the stack vocabulary
- δ is the transition function
- s is the start state
- γ is the stack start symbol, $\gamma \in \Gamma \cup \{\epsilon\}$
- F is a set of final / accepting states

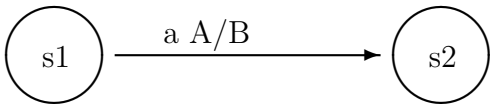
$\delta : (S \times (V \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\})) \mapsto (S \times (\Gamma \cup \{\epsilon\})$

Note that by convention we will use upper case letters to represent words from the stack vocabulary, and lower case letters for words from the input vocabulary (cf. the similar convention for non-terminal and terminal symbols in grammars).

Here δ is a function from states, input symbols and stack symbols, to states and stack symbols. For example if

$$\delta(s_1, a, A) = \langle s_2, B \rangle$$

this means that if you are in state s_1 looking at input symbol a, and are able to pop A off the stack, then you can move to state s_2 and push B onto the stack. In graphical form, this would correspond to



As with finite state machines, it is not necessary for δ to be a function. It can be a relation, or equivalently, a function onto sets of state / stack symbol pairs, e.g.

$$\delta(s_1, a, A) = \{\langle s_2, B \rangle, \langle s_3, C \rangle, \langle s_2, C \rangle, \langle s_3, B \rangle\}$$

In this case, the PDA is non-deterministic: from a single configuration of state, input symbol and stack top, multiple transitions are possible.

6.1.3 Languages Accepted by PDAs

A *PDA Configuration* is a triple $\langle s, i, \alpha \rangle$
where

- s is the current state
- i is the remaining input
- α is the stack

Taking a transition moves you from one configuration to another

$\langle s_1, i, \alpha \rangle \vdash \langle s_2, j, \beta \rangle$
— one transition takes you from $\langle s_1, i, \alpha \rangle$ to $\langle s_2, j, \beta \rangle$

$\langle s_1, i, \alpha \rangle \overset{*}{\vdash} \langle s_2, j, \beta \rangle$
— zero or more transitions take you from $\langle s_1, i, \alpha \rangle$ to $\langle s_2, j, \beta \rangle$

(Here we use $\vdash$ to represent transitions between PDA configurations.) Thus, for example, if we have the transition

$$oprs_2, B \in \delta(s_1, a, A)$$

then the following is a possible transition between configurations:

$\langle s_1, au, A\alpha \rangle \vdash \langle s_2, u, B\alpha \rangle$

A string $u \in V^*$ is accepted by a $PDA = \langle S, V, \Gamma, \delta, s, \gamma, F \rangle$ iff

$\langle s, u, \gamma \rangle \overset{*}{\vdash} \langle s_f, \epsilon, \epsilon \rangle$

— where $s_f \in F$

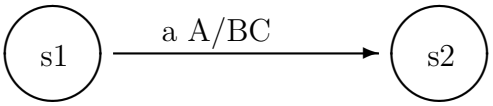
In other words, we have to be able to take a sequence of transitions, beginning at the start state with the entire string and the stack initialised to contain just its start symbol, and ending in a final state with no remaining input and an empty stack.

6.1.4 Variations on PDAs

There are a number of variants on PDAs, all of which can be used to accept the same class of languages as the kind of machine we have seen above.

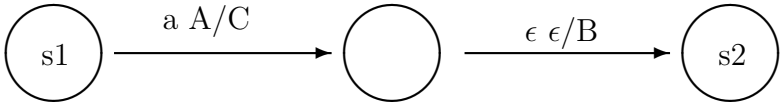
Extended PDAs

An extended PDA allows you to push a string of symbols onto the stack, rather than just a single symbol. For example, one might have an extended transition



One taking a transition from s_1 to s_2, this pops the single symbol A from the stack, but pushes two symbols, BC onto it.

One can readily convert an extended PDA to a non-extended one by adding extra intermediate states connected by empty transitions that just push one symbol at a time, e.g.



Acceptance Conditions

Previously we defined acceptance as reaching a final state with no remaining input and an empty stack. A *final state* PDA is one that accepts a string if it reaches a final state with no remaining input, even if there is still something on the stack. An *empty stack* PDA is one that accepts if it reaches any state with no remaining input and an empty stack.

An empty stack PDA is just an ordinary PDA where all states are final states. A final state PDA can be converted to an ordinary one by having empty transitions looping around on final states that pop off any remaining stack symbols.

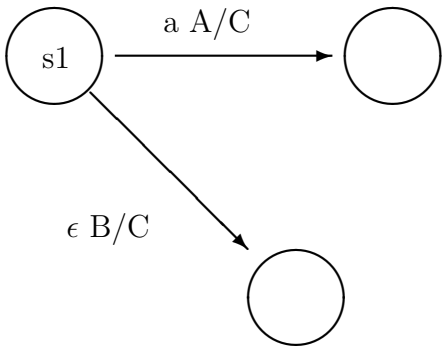
One sometimes also finds stacks defined in such a way that stack operations can only be performed on non-empty stacks (i.e. you can't even pop ϵ off an empty stack). Execution halts whenever the stack is emptied. In this case, the stack start symbol γ obviously cannot be ϵ.

6.1.5 Determinism and PDAs

Unlike finite state automata, there is a difference between the class of languages accepted by deterministic and non-deterministic PDAs. Or put another way, there is no method for converting any non-deterministic PDA to a deterministic PDA accepting the same language.

Non-deterministic PDAs define the wider class of languages: the context free languages. Deterministic PDAs define *deterministic languages*.[1]

Recall that a PDA is deterministic if for any state / input / stack combination, more than one transition is possible. Different transitions may move to different states, or to the same states but pushing different symbols onto the stack. The presence of an empty transition (one that does not consume any input) does not necessarily make a PDA non deterministic. For example, the following (part of a) PDA introduces no non-determinism: the empty transition and the a-transition are applicable only to distinct stack configurations



Non-determinism in the algorithm for acceptance by a PDA is handled in the same way as for a non-deterministic finite state automaton. That is, a stack is used to record choice points. When execution blocks in a non-terminating configuration, a choice point is popped from the stack, and the algorithm backtracks to that point.

[1] Deterministic languages are lie between regular and context free languages. However, they are not an especially important class and are usually lumped under the heading of context free.

6.2 From CFGs to PDAs

We now show that context free grammars and non-deterministic PDAs define the same class of languages. This is done in two parts. In this section we show how to convert a context free grammar to a PDA. And in the next section we show how to convert a PDA to a context free grammar.

6.2.1 Conversion from Grammars in GNF

The basic idea behind converting a CFG to a PDA is to equate

- Stack vocabulary = non-terminal symbols
- Input vocabulary = terminal symbols

The idea is that a rule (in Greibach Normal Form) like

$$A \to aBC$$

should be seen as an instruction to take an a-transition, popping A off the stack, and pushing BC onto the stack in its place.

More generally, let G be a grammar in Greibach normal form. That is all rules take either the form

$$A \to aA_1 \dots A_n$$

(where $A, A_1, \dots A_n$ are terminal symbols, $n \geq 0$ and a is a terminal symbol), or the form

$$S \to \epsilon$$

(where S is the start symbol).

(Extended) PDA for GNF grammar, $G = \langle N, V, R, S\rangle$:

$PDA = \langle Q, V, \Gamma, \delta, s_0, \epsilon, F\rangle$
where

$Q = \{s_0, s_1\}$
$\Gamma = N - \{S\}$
$F = \{s_1\}$

and with the transitions

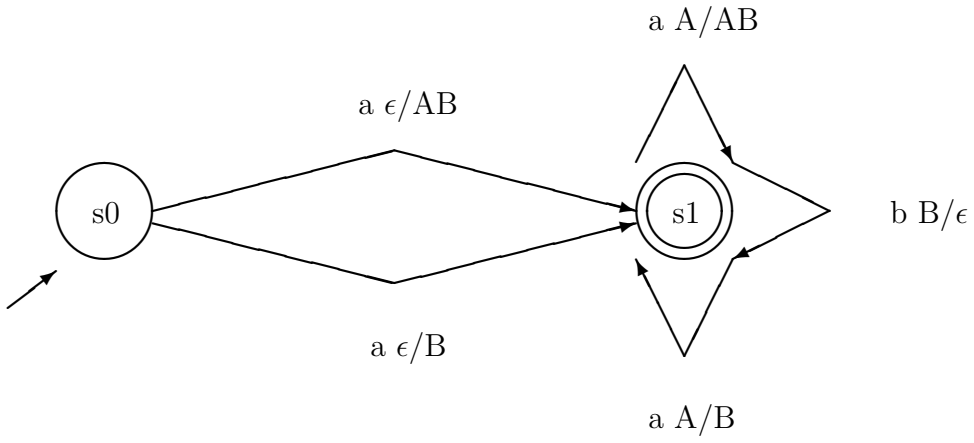
1. $\delta(s_0, a, \epsilon) = \{\langle s_1, U\rangle \mid (S \to aU) \in R\}$
   (where U is a possibly empty sequence of non-terminals)
2. $\delta(s_1, a, A) = \{\langle s_1, U\rangle \mid (A \to aU) \in R \text{ and } A \in \Gamma\}$
   (for each non-terminal, A).
3. $\delta(s_0, \epsilon, \epsilon) = \{\langle s_1, \epsilon\rangle\}$ if $S \to \epsilon \in R$

6.2.2 Example: PDA for $a^n b^n$

To illustrate this conversion, consider the GNF grammar for $a^n b^n$:

$S \to aAB \mid aB$
$A \to aAB \mid aB$
$B \to b$

The PDA corresponding to this, in graphical form, is



To see how the PDA relates to the grammar, consider the leftmost derivation of the string *aaabbb* in the grammar:

Sentential form	Rule
S	$S \rightarrow aAB$
aAB	$A \rightarrow aAB$
$aaABB$	$A \rightarrow aB$
$aaaBBB$	$B \rightarrow b$
$aaabBB$	$B \rightarrow b$
$aaabbB$	$B \rightarrow b$
$aaabbb$	

If you look at the successive sentential forms, they consist of a terminal prefix and a non-terminal suffix. At any point, the terminal prefix can be viewed as the input string so far processed, and the non-terminal suffix as a stack of symbols that need to be popped off. Each rule application pops one stack symbol off, but may push further stack symbols on.

Compare the derivation in the grammar to the transitions between PDA configurations for the same string:

$\langle s_0, aaabbb, \epsilon\rangle \vdash$
$\langle s_1, aabbb, AB\rangle \vdash$
$\langle s_1, abbb, ABB\rangle \vdash$
$\langle s_1, bbb, BBB\rangle \vdash$
$\langle s_1, bb, BB\rangle \vdash$
$\langle s_1, b, B\rangle$
$\langle s_1, \epsilon, \epsilon\rangle$

Note how at each step, the stack is the same as the non-terminal suffix in the grammar derivation. And the initial input minus the remaining input is the same as the terminal prefix.

The similarity between the leftmost derivation and the sequence of transitions forms the basis for an inductive proof that the conversion procedure generates the same language. But we will not go into this proof.

6.2.3 Conversion from Grammars not in GNF

It is not necessary to convert a grammar to Greibach normal form before converting it to a PDA. However, in this case, the stack and input vocabulary overlap. Also, if the grammar contains any left recursion, there is no guarantee that the PDA will always terminate when trying to accept or reject a string.

PDA from grammar $G = \langle N, V, R, S \rangle$:

$PDA = \langle Q, V, \Gamma, \delta, s_0, \gamma, F \rangle$
where

$Q = \{s_0, s_1, s_2\}$
$\Gamma = N \cup V \cup \{\gamma\}$
$F = \{s_2\}$

and with the transitions

1. $\delta(s_0, \epsilon, \gamma) = \{\langle s_1, S\gamma \rangle\}$
2. $\delta(s_1, \epsilon, A) = \{\langle s_2, U \rangle \mid (A \rightarrow aU) \in R\}$
   (for each non-terminal, A)
3. $\delta(s_1, a, a) = \{\langle s_1, \epsilon \rangle\}$
   (for each terminal, a)
4. $\delta(s_1, \epsilon, \gamma) = \{\langle s_2, \epsilon \rangle\}$

Thus, if we take the grammar

$$S \rightarrow aSb \mid ab$$

we get the PDA with the transitions

$\delta(s_0, \epsilon, \gamma) = \{\langle s_1, S\gamma \rangle\}$
$\delta(s_1, \epsilon, S) = \{\langle s_1, aSb \rangle, \langle s_1, ab \rangle\}$
$\delta(s_1, a, a) = \{\langle s_1, \epsilon \rangle\}$
$\delta(s_1, b, b) = \{\langle s_1, \epsilon \rangle\}$
$\delta(s_1, \epsilon, \gamma) = \{\langle s_2, \epsilon \rangle\}$

6.3 From PDAs to CFGs

The conversion from a PDA to a CFG is fiddly, and is not the sort of thing you would be expected to do in an exam.

Recall that in converting FSMs to Regular Grammars, the input vocabulary became the set of terminals, the states became the non-terminals, and the transitions defined the rules. Similarly with PDAs, except that the non-terminals are triples $\langle s_i, A, s_j \rangle$, where s_i and s_j are states, and A a stack symbol. There is in addition a start symbol, S

The triple $\langle s_i, A, s_j \rangle$ represents a sequence of transitions starting in state s_i with A on the top of the stack, and ending in state s_j, with the stack the same as before except that A has been popped. Note that the sequence of transitions may push and pop a number of intermediate values on the stack; but we are only

concerned with what is on the stack at the start and the end of the transition sequence.

Assume a $PDA = \langle Q, V, \Gamma, \delta, s_0, \gamma, F \rangle$ Add extra transitions to it as follows

- If $\langle s_j, \epsilon \rangle \in \delta(s_i, u, \epsilon)$, add
  $\delta(s_i, u, A) = \{\langle s_{,j}\, , A \rangle \mid A \in \Gamma\}$
- If $\langle s_j, B \rangle \in \delta(s_i, u, \epsilon)$, add
  $\delta(s_i, u, A) = \{\langle s_{,j}\, , BA \rangle \mid A \in \Gamma\}$

Derive rules from transitions

1. For each $s_j \in F$, there is a rule
   $$S \rightarrow \langle s_0, \gamma, s_j \rangle$$
2. For each transition $\langle s_j, B \rangle \in \delta(s_i, x, A)$ (where $A \in \Gamma \cup \{\epsilon\}$), and for each state $s_k \in Q$, there is a rule
   $$\langle s_i, A, s_k \rangle \rightarrow x \ \langle s_j, B, s_k \rangle$$
3. For each transition $\langle s_j, BA \rangle \in \delta(s_i, x, A)$ (where $A \in \Gamma$), and for each state $s_k, s_n \in Q$, there is a rule
   $$\langle s_i, A, s_k \rangle \rightarrow x \ \langle s_j, B, s_n \rangle \ \langle s_n, A, s_k \rangle$$
4. For each state $s_k \in Q$, there is a rule
   $$\langle s_k, \epsilon, s_k \rangle \rightarrow \epsilon$$

Explanations of these derived rules are as follows:

1. $S \rightarrow \langle s_0, \gamma, s_j \rangle$

   This says that to find an S, you must start in the start state, and find some way to reach a finish state that removes the initial stack symbol from the top of the stack

2. $\langle s_j, B \rangle \in \delta(s_i, x, A)$ gives rise to
   $\langle s_i, A, s_k \rangle \rightarrow x \ \langle s_j, B, s_k \rangle$

   There is a transition from s_i that consumes x, replaces A by B on the top of the stack, and moves you to state s_j.

   The rule says, suppose you are in s_i, looking to remove A from the stack (and not replace it with anything else), and want to reach s_k.

   Then consume an x, replace A on the stack by B, and move to s_j.

   To reach s_k you have to find another rule that will take you from s_j to s_k and remove the B (without replacing it by anything else).

3. $\langle s_j, BA \rangle \in \delta(s_i, x, A)$ gives rise to
   $\langle s_i, A, s_k \rangle \rightarrow x \ \langle s_j, B, s_n \rangle \ \langle s_n, A, s_k \rangle$

There is a transition from s_i that consumes x, replaces A by BA on the top of the stack, and moves you to s_j.

The rule says, suppose you are in s_i, looking to remove A from the stack (and not replace it with anything else), and want to reach s_k.

Then consume an x, replace A by BA, and move to s_j.

From s_j, you then have to find a rule that will move you to some other state s_n and completely remove the B from the stack.

From s_n you have to find another rule that will move you to s_k and completely remove the A from the stack.

4. $\langle s_k, \epsilon, s_k \rangle \rightarrow \epsilon$

   Represents a transition from a state to itself that neither consumes input nor alters the stack.

   These rules are used to terminate derivations.

As with the conversion from CFGs to PDAs, the idea is to construct a CFG whose leftmost derivations simulate the sequence of transitions in the PDA

6.4 Summary

This chapter has introduced push-down stack automata, which are finite state automata supplemented with a stack like memory.

Unlike finite state automata, deterministic and non-deterministic PDAs define different classes of languages. Non-deterministic PDAs define context free languages. This is shown by giving procedures for converting a CFG into a non-deterministic PDA, and vice versa.

Chapter 7

Parsing

This chapter deals with the parsing of context free languages. This is a topic of considerable practical importance, since a central part of any compiler for a (context free) programming language is a parser. A *parser* takes a string and a grammar, and returns derivation trees for the string, if any exist. In program compilation, the derivation tree is used to construct a segment of machine code corresponding to the meaning of the program parsed.

A parser differs from a *recogniser* for a language. A recogniser — such as a finite state automaton for a regular language or a non-deterministic push-down stack automaton for a context free language — just takes a string and returns a yes-or-no answer, depending on whether the string is a member of the language or not. A parser both gives a yes-no-answer, and if the string is a member of the language gives its derivation within a particular grammar.

The relativisation of a parser to a grammar is important. As we have already seen (e.g. in converting CFGs to Chomsky or Greibach normal form), the same language may be defined by a number of different grammars. But these grammars will typically give rise to different derivations for strings in the language. In other words, derivation trees are relative to grammars, rather than relative to languages.

The chapter is organised as follows. Section 7.1 explains why derivation trees are useful, e.g. in program compilation. Section 7.2 discusses non-deterministic top-down parsing of context free languages, using either depth-first or breadth-first search. Section 7.3 discusses non-deterministic bottom-up (or shift-reduce) parsing, again either with depth-first or breadth first search. Section 7.4 discusses deterministic top-down, or LL(k), parsing. And Section 7.5 discusses deterministic bottom-up, or LR(k), parsing.

7.1 Derivation and Meaning

By and large in this course we have ignored questions of what sentences in a formal language might mean. Instead, we have been concerned with rather

narrower issues, such as what strings of words are and are not sentences, and how this can be decided by grammars or automata. This is about the one place in the course where we will look at what sentences mean, and how this relates to other topics covered in formal language theory. This is of relevance not only to program compilation, but also to computational attempts to process and understand natural languages.

7.1.1 The Meaning of Arithmetic Expressions

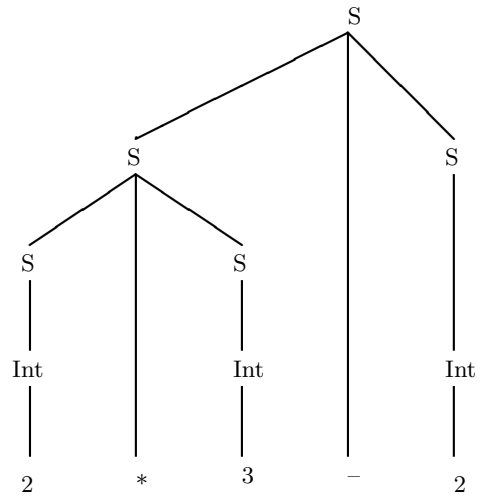
Consider the following grammar for arithmetic expressions, such as as $2 * 3 - 2$

$$S \rightarrow S + S \mid S - S \mid S * S \mid S/S \mid Int$$
$$Int \rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

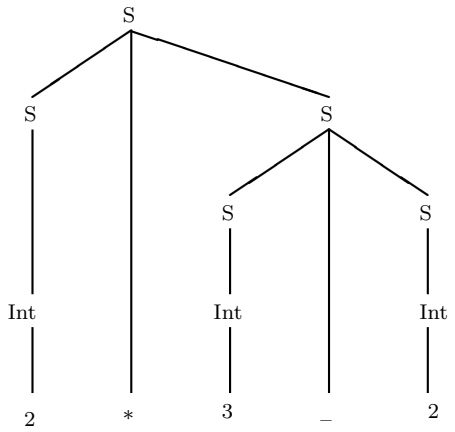
Here, S is the start symbol, and the rules allow it to be rewritten to any arithmetic expression, including single integers.

It is not difficult to show that the grammar given above is ambiguous. There are two alternative derivation trees for the string $2 * 3 - 2$, for example:

- $(2 * 3) - 2$:



- $2 * (3 - 2)$:



The two derivations correspond to two different meanings of the string: $(2 * 3) - 2 = 4$ and $2 * (3 - 2) = 1$.

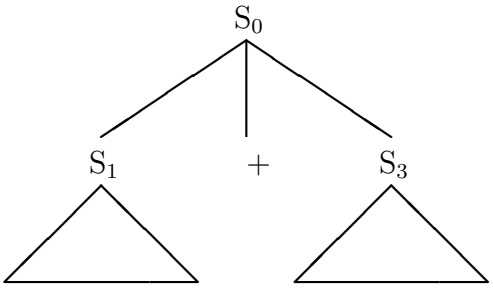
We can associate meanings with (derivations) of strings by pairing meaning (or semantic) rules with the individual grammar rules, e.g.

Grammar Rule	Semantic Rule
$S \rightarrow S + S$	$\mathbf{S}_0 = eval(\mathbf{S}_1 + \mathbf{S}_3)$
$S \rightarrow S - S$	$\mathbf{S}_0 = eval(\mathbf{S}_1 - \mathbf{S}_3)$
$S \rightarrow S * S$	$\mathbf{S}_0 = eval(\mathbf{S}_1.\mathbf{S}_3)$
$S \rightarrow S/S$	$\mathbf{S}_0 = eval(\mathbf{S}_1/\mathbf{S}_3)$
$S \rightarrow Int$	$\mathbf{S}_0 = \mathbf{Int}_1$
$Int \rightarrow 1$	$\mathbf{Int}_0 = \mathbf{1}$
$Int \rightarrow 2$	$\mathbf{Int}_0 = \mathbf{2}$
...	
$Int \rightarrow 9$	$\mathbf{Int}_0 = \mathbf{9}$

The semantic rule for the first grammar rule

$$\mathbf{S}_0 = eval(\mathbf{S}_1 + \mathbf{S}_3)$$

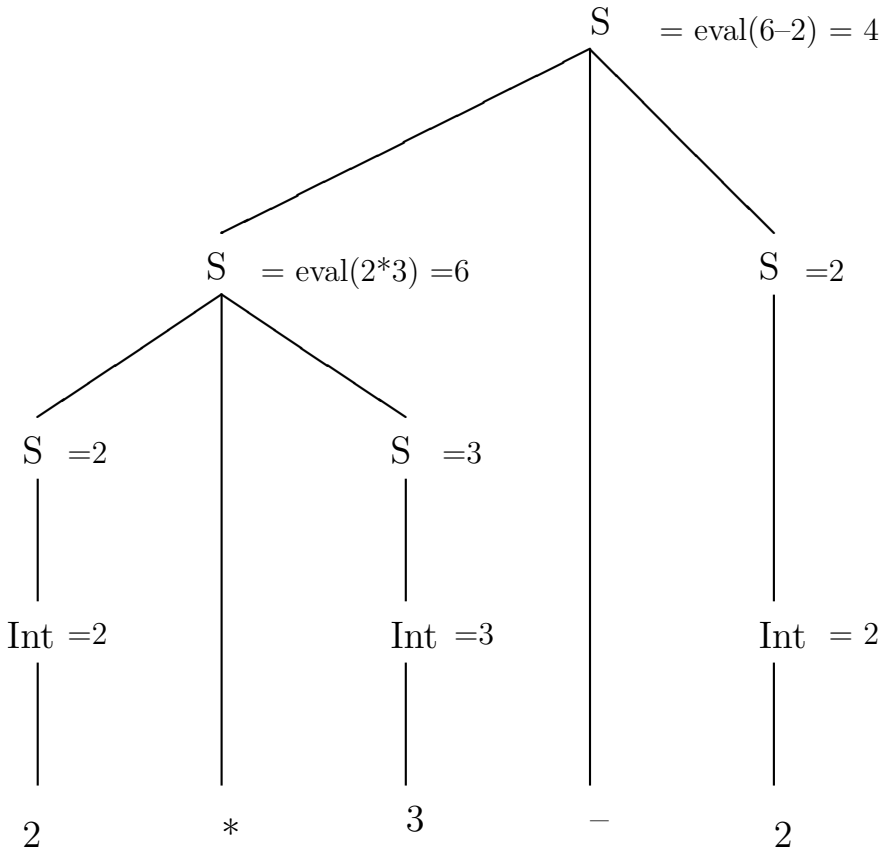
is to be understood as follows. Suppose that the rule is used iat some point in a derivation to construct a tree S_0 having three subtrees S_1, + and S_3 as shown



The meaning of the tree S_0, which we represent in **boldface** as $\mathbf{S}_0$ is obtained by taking the meanings of its two S subtrees, $\mathbf{S}_1$ and $\mathbf{S}_3$ and evaluating what you get by adding them together.

Thus, for example, if $\mathbf{S}_1 = \mathbf{4}$ and $\mathbf{S}_3 = \mathbf{2}$, then $\mathbf{S}_0 = eval(\mathbf{4} + \mathbf{2}) = \mathbf{6}$

To construct the meaning for a string like $2 * 3 - 2$, we take a derivation tree and build up the meaning of the root node recursively from that of its subtree. For the first derivation tree for the string, this gives



where the meanings associated with nodes in the tree are written on the right hand sides of the equality signs.

Exercise: show how the second derivation tree for $2 * 3 - 2$ assigns the string a meaning of 1.

7.1.2 Parsing and Meaning

The preceding example about arithmetic expressions illustrates how the way that a string is derived determines its meaning. Indeed, it shows how an ambiguous string having more than one derivation tree can thereby have more than one meaning.

In order to determine the meaning of a string, given some grammar for which semantic rules are defined, it is necessary to determine what the derivation tree(s) for the string is(are). And this is what parsers are for.

This is why parsing is such an important part of program compilation. Programming languages are defined by (unambiguous) context free grammars, where

each grammar rule has an associated semantic rule. The semantic rules build up chunks of machine code for a node in a derivation tree by combining the machine code of its daughter nodes. But in order to do this, a parser first needs to construct the derivation tree.

A similar approach is applied in natural language processing. Again there is a grammar with associated semantic rules, and sentences have to be parsed before their meanings can be constructed. In the case of language processing, however, the semantic rules normally assign bits of logical expressions to nodes in the derivation tree.

(One significant difference between parsing programming languages and natural languages is that grammars for natural languages are typically highly ambiguous, whereas grammars for programming are deliberately designed to be unambiguous. Why are grammars for programming languages unambiguous?)

Having said why parsing is a useful thing to do, we now turn to the question of how to do it.

7.2 Top-Down Parsing

The simplest way of parsing a string would be to do repeated (leftmost) derivations in the grammar, until eventually one derives the string that you are trying to parse. To a first approximation, this is what a top-down parser does.

Top down parsing begins with a sentential form comprising the start symbol of the grammar. Then a grammar rule is chosen to expand the leftmost non-terminal in the sentential form. This is repeated until the sentential form is expanded out into the string to be parsed. As the sentential form is expanded, a derivation tree is constructed in parallel: each time a rule expands a non-terminal, daughters are added to the corresponding non-terminal in the derivation tree.

7.2.1 Top-Down Parsing Ignoring Non-Determinism

If rules expanding the sentential form are chosen blindly, it will be pure luck if the first terminal string that the sentential is expanded out to is the string that we want to parse. We can fudge the issue by assuming the existence of a procedure *choose-correct-rule*, which always selects a rule that takes us towards the required terminal string. (In general it is not possible to define such a procedure, though it is possible for a special class of context free grammars discussed in section 7.4).

We can also ensure early failure by checking to see if the terminal prefix of the expanded sentential form is the same as the prefix of the terminal string to be parsed. If not, we can stop parsing before going all the way to producing a complete sentence distinct from the one we want to parse.

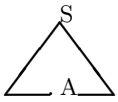
An algorithm for top-down parsing, assuming the existence of a procedure

choose-correct-rule is shown below:

Outline algorithm for top down parsing of a string p:

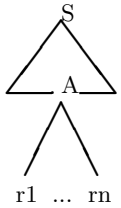
top-down-parse(p):

1. SentForm := S; Tree := S;
2. REPEAT until SentForm == p
   (a) SentForm==uAv, where A is the leftmost non-terminal;

   Tree== 

   (b) If u is not a prefix of p, then RETURN FALSE;
   (c) If *choose-correct-rule*$(uAv) = A \rightarrow r_1 \ldots r_n$,
   Then

   SentForm := $ur_1 \ldots r_n v$;

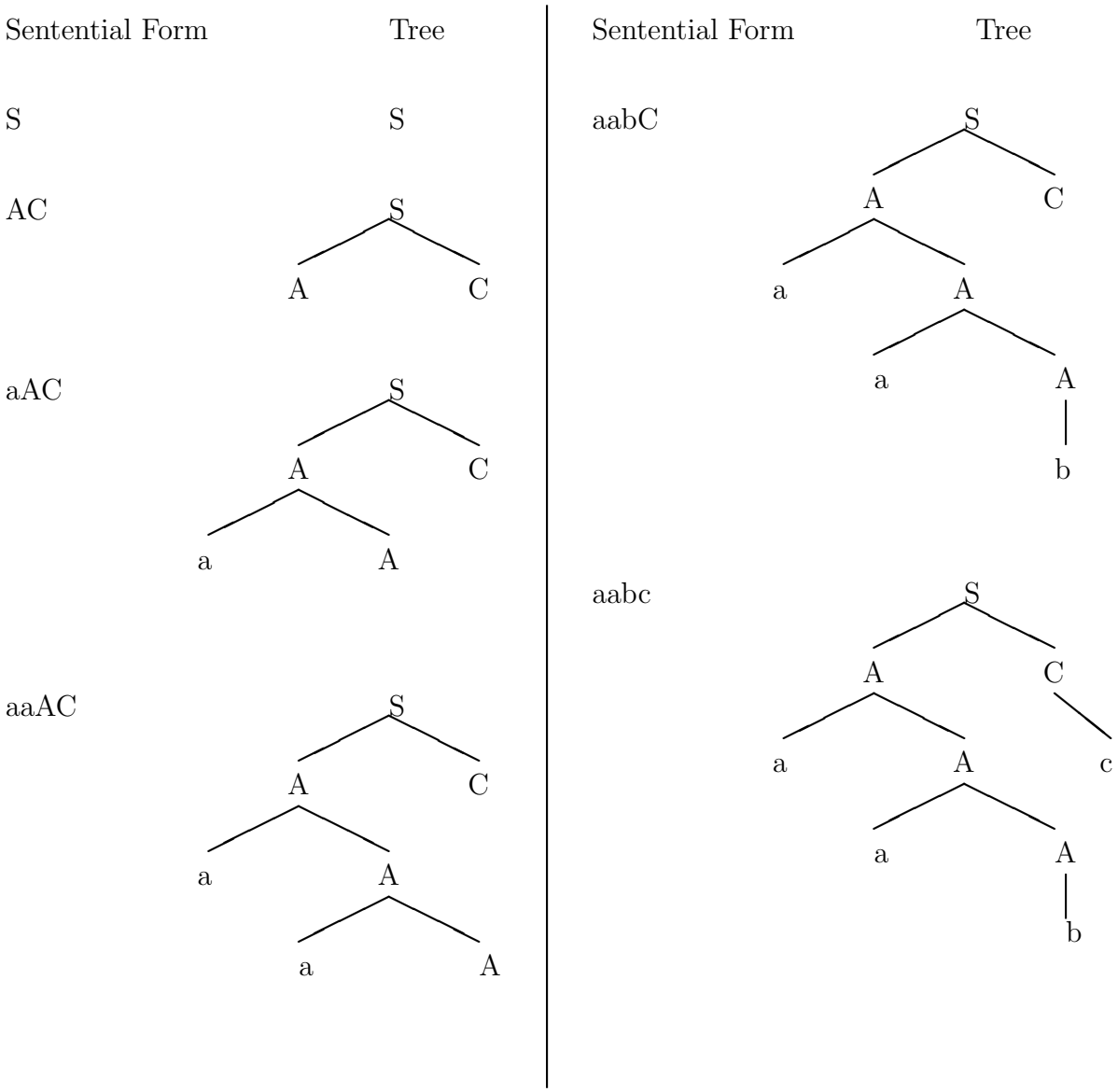
   Tree := 

   Else RETURN FALSE
3. RETURN Tree

To see how it works, suppose we have the grammar

$$S \rightarrow AC$$
$$A \rightarrow aA \mid b$$
$$C \rightarrow cC \mid c$$

and that we want to parse the string *aabc*. By choosing the correct rule at each step ($S \rightarrow AC$, $A \rightarrow aA$, $A \rightarrow aA$, $A \rightarrow b$, $C \rightarrow c$), the parsing algorithm expands the sentential form and builds up the tree as shown:



Note, however that if at the second step we had instead used the rule $A \rightarrow b$ to expand the A in the sentential form AC, parsing of *aabc* would have failed. It would have failed even though there is a derivation of the string *aabc*

7.2.2 Non-Deterministic Top-Down Parsing

The algorithm just shown is one for deterministic top-down parsing. But in general parsing is a highly non-deterministic operation: there is usually no way of telling which is the correct grammar rule to apply at any one point. In other words, for most (though not all) grammars it is impossible to define a procedure like *choose-correct-rule*. Therefore, parsing algorithms have to account for the fact that the wrong rule may sometimes be chosen (e.g. choosing $A \rightarrow b$ instead of $A \rightarrow aA$ in the example above). When a wrong rule has been chosen, leading

to a failure to derive the required string, the algorithm must be able to retrace its steps and try using an alternative rule.

The situation is analagous to that for non-deterministic finite state automata or push-down stack automata. These use a stack to record previous choice points, and backtrack by popping choice points off the stack. In parsing, using a stack to record choice points leads to something known as *top-down* parsing. An alternative regime is possible, however. This uses a queue[1] instead of a stack, and leads to *breadth-first* parsing.

The following is an algorithm for non-deterministic top-down parsing. It can be adapted to give either depth-first or breadth-first parsing, depending on whether a stack or queue is used to record choice points, ⟨Sentential Form, Tree⟩.

[1] A queue is a first-in, first-out data structure: new items are added to the end of the queue, and old items are removed from the front of the queue.

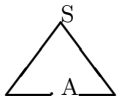
Non-Deterministic Top-Down Parser:

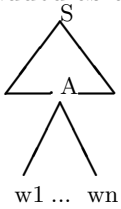
parse(p)

1. SentForm := S; Tree := S;
   StackOrQueue := empty;
   ADD($\langle$SentForm, Tree$\rangle$, StackOrQueue);
2. REPEAT UNTIL SentForm= p or StackOrQueue is empty
   (a) REMOVE($\langle$SentForm, Tree$\rangle$, StackOrQueue);
   (b) Let SentForm == uAv,
   where A is the leftmost non-terminal;
   (c) IF u is a (terminal) prefix of p
   THEN FOR EACH rule $A \rightarrow w$
   ADD($\langle uwv,$ (Tree + $w)\rangle$, StackOrQueue);
3. If SentForm= p, then RETURN Tree;

Notes
If StackOrQueue is a stack:
ADD = PUSH, REMOVE = POP,
Gives depth-first search
If StackOrQueue is a queue:
ADD = ENQUEUE, REMOVE = DEQUEUE,
Gives beadth-first search

Tree+w indicates tree with w added as daughters of A

Tree = 

Tree+w = 

Breadth-First, Depth-First and Graphs of Grammars

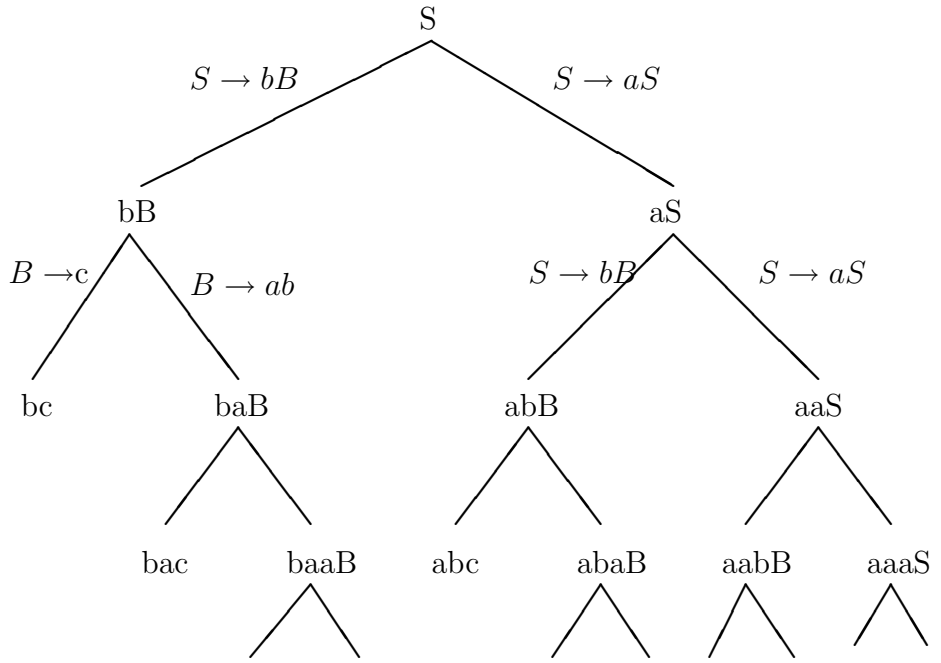
The terms depth-first and breadth-first refer to different strategies for handling *search problems*. A search problem involves looking through a space of possibilities to find a particular item. The space of possibilities can often be represented as a *search tree*. To find a particular item in the search tree, you start at the root node of the tree, and start descending through the branches of the tree until find some item you want.

In the case of parsing, we are looking through the space of all possible derivations in the grammar in order to find one or more derivations of the string to be parsed. The *graph* of a grammar shows all possible leftmost derivations, in

a particular order. Consider the grammar:

$S \rightarrow bB \mid aS$
$B \rightarrow c \mid aB$

We can draw a graph for the grammar showing all possible (leftmost) derivations:[2].



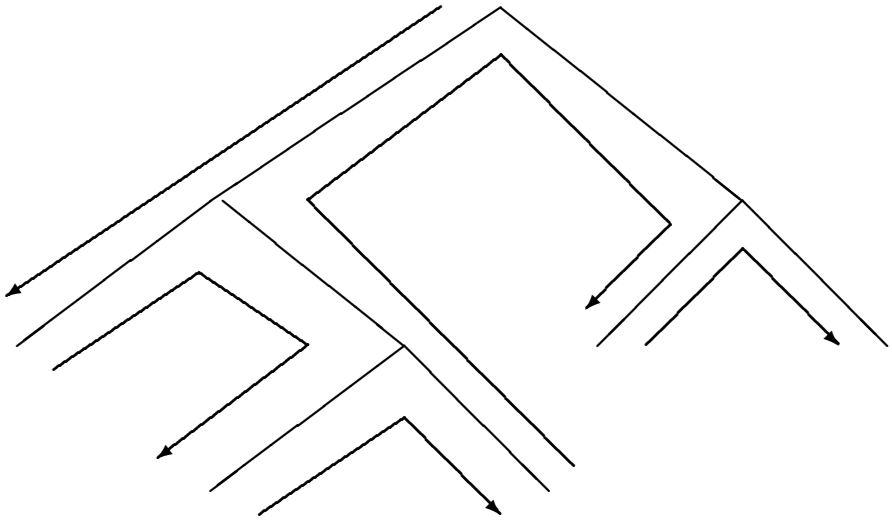
Any path from the root of the graph to a leaf node corresponds to a leftmost derivation of the string labelling the leaf node.

Depth-first and breadth-first search correspond to different ways of looking through the search tree (provided by the grammar graph) for terminal strings. Depth-first search explores each path from the root to a leaf node in turn. It descends first to the leftmost leaf node, then retraces its steps back a level and descends to the next leaf node, and so on

[2]The graph is constructed as follows. First, take the start symbol as the root node. Then repeatedly expand each node containing a non-terminal symbol as follows:

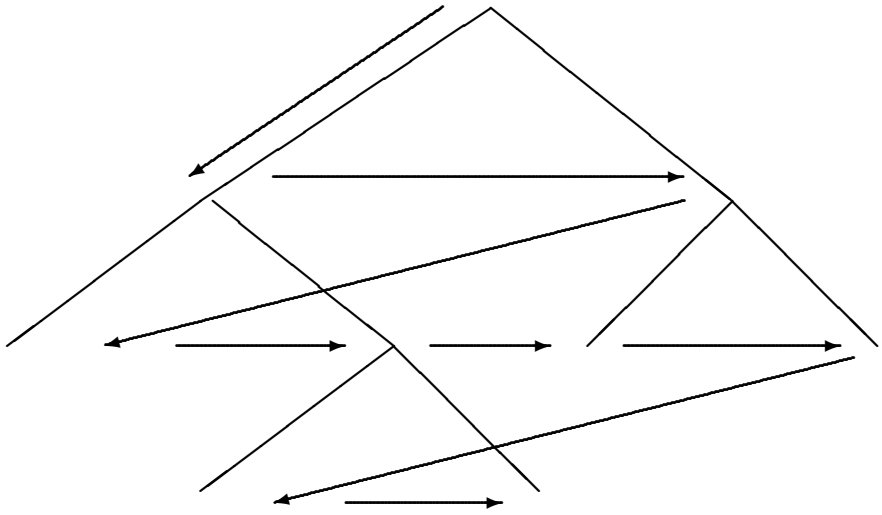
- Take the leftmost non-terminal in the string written at the node
- For each rule expanding the leftmost non-terminal
  - Add a branch beneath the node
  - Expand the non-terminal using the rule, and add the resulting sentential form as a daughter

Repeat for as long as there are nodes containing non-terminal symbols. Obviously, for grammars defining infinite languages, the graph of the grammar is infinite



Depth-first search

Breadth-first search explore all the paths from root to leaves in parallel. It hops between branches on the search tree, not descending to the nth level in the graph until it has looked at all strings at the $(n-1)$th level



Breadth-first Search

Breadth-first search can be likened to pouring ink in at the top (root node) of the search tree, and watching it spread down level by level. Depth-first search can be likened to turning the search on its side, and pouring ink in through the root node. Here the ink will spread to the lowest (i.e. leftmost) leaf node before moving up to the next leaf node.

For top-down parsing, depth-first and breadth-first search have the following relative advantages and disadvantages:

Depth-First

- **Against:** Grammars for infinite languages will have some infinitely descending branches in the graph

  Depth first search can get stuck in these infinite branches, and never get back to search the rest of the graph

- **Against:** In other words, depth-first search can (a) fail to terminate, and (b) is not guaranteed to find a parse even if one exists.

- **For:** Depth first search will usually find an analysis, if there is one, quicker than breadth-first.

  (Unless of course it fails to terminate because of hitting an infinite branch)

- **Uses:** Most useful if you want to find the first (of possibly many) parses of a sentence

Breadth-First

- **Against:** Because it is building up all analysis in parallel, it can be computationally more expensive than depth-first

- **For:** Infinite branches do not cause non-termination and failure to explore all of the search space.

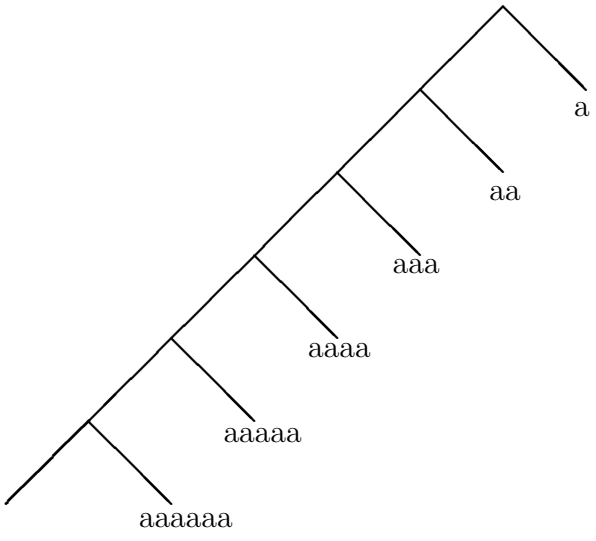
- **For:** In other words, if there is a parse it will (eventually!) find it

- **Uses:** Most useful if you want to find all possible parses for a sentence

The potenital non-termination of top-down depth-first parsing is especially critical if the grammar contains any left-recursive rules. Consider the graph for the following grammar for the language a^+

$S \rightarrow Sa \mid a$

Graph for grammar:



Here, the leftmost branch is infinitely descending (the leftmost branch because of the left recursive rule). Any attempt to search the graph depth-first will therefore get caught up in trying to pursue the leftmost branch to its end, and will never backtrack out of this in order to find the terminal strings on other leaf nodes.

7.3 Bottom-Up Parsing

In top-down parsing, the idea is to begin with the start symbol of the grammar, and repeatedly apply grammar rules to expand out sentential forms until you get to the terminal string you want. In bottom-up parsing, the idea is to start with the terminal string and repeatedly apply grammar rules in the opposite direction to reduce the string, until eventually it is reduced to the start symbol of the grammar.

To illustrate the difference, let us look again at the grammar we used to illustrate top-down parsing:

$$S \rightarrow AC$$
$$A \rightarrow aA \mid b$$
$$C \rightarrow cC \mid c$$

and suppose that we want to parse the string $aabc$.

The top-down parse will apply rules top-down to expand out non-terminal symbols as follows:

Top-Down

String	Rule
$\underline{S}$	$S \rightarrow AC$
$\underline{A}C$	$A \rightarrow aA$
$a\underline{A}C$	$A \rightarrow aA$
$aa\underline{A}C$	$A \rightarrow b$
$aab\underline{C}$	$C \rightarrow c$
$aabc$	

The bottom-up parse will apply rules in the opposite direction to reduce (sub)sequences of symbols into single non-terminal symbols:

Bottom-Up

String	Rule
$aab\underline{c}$	$C \rightarrow c$
$aa\underline{b}C$	$A \rightarrow b$
$a\underline{a}AC$	$A \rightarrow aA$
$\underline{a}AC$	$A \rightarrow aA$
$\underline{AC}$	$S \rightarrow AC$
S	

7.3.1 Shift-Reduce Parsing

The reverse order, bottom-up derivation for the string *aabc* shown above does not quite accurately reflect the way in which bottom-up parsing is normally carried out. The standard algorithm is *shift-reduce* parsing. Before formally describing the algorithm, we will introduce it by means of an example showing how the two basic operations of *reduce* and *shift* work.

Reducing is the process of taking a string of symbols matching the right hand side of a grammar rule, and reducing them to the single symbol on the left hand side of the rule. Hence the following is a reduction via the rule $A \rightarrow aA$

$$a\underline{aA} \overset{A \rightarrow aA}{\Longrightarrow} aA$$

The main part of bottom-up parsing is to repeatedly reduce strings of symbols in this way.

Shifting comes into play as follows. The parser maintains the string to be parsed in two parts: a reduced part and an unreduced part The first part only is subject reduce operations. The second, unreduced part consists only of terminal symbols. A terminal symbol may be shifted, one at a time, from the start of the undreduced part onto the end of the reduced part.

Shift-reduce parsing starts by placing the entire string to be parsed into the unreduced part. Terminal symbols are shifted, one at a time, onto the end of the (initially empty) reduced part. Reductions are carried out only on the final suffixes of the reduced string.

Here, for example, is a shift reduce parse of the string *aabc* using the grammar shown previously, showing the order in which shift and reduce operations are interleaved.

Reduced	Unreduced	Operation
ϵ	$aabc$	Shift
a	abc	Shift
aa	bc	Shift
aab	c	Reduce: $A \rightarrow b$
aaA	c	Reduce: $A \rightarrow aA$
aA	c	Reduce: $A \rightarrow aA$
A	c	Shift
Ac	ϵ	Reduce: $C \rightarrow c$
AC	ϵ	Reduce: $S \rightarrow AC$
S	ϵ	Parsed

Note how reductions only ever apply to the final part of the reduced string. Parsing terminates successfully if the reduced string comprises just the start symbol of the grammar, and the unreduced string is empty.

7.3.2 Non-Deterministic Shift-Reduce Parsing

As with top-down parsing, bottom-up (shift-reduce) parsing is usually non-deterministic: it is not normally possible to correctly decide at each point what should be done — either shift or reduce, and if reduce, with which rule.

This non-determinism manifests itself in two types of conflict

- Shift-reduce conflicts:
  These occur when there is a choice between reducing of or shifting the next non-terminal symbol.
- Reduce-reduce conflicts:
  These occur when there is a choice of reducing using two or more different rules.

The following grammar can be used to illustrate these conflicts:

$S \rightarrow aA \mid aAA \mid C$
$A \rightarrow b$
$C \rightarrow Ab$

Consider parsing the string *abb*:

Reduced	Unreduced	Operation
ϵ	abb	Shift
a	bb	Shift
ab	b	Reduce: $A \rightarrow b$
		shift-reduce conflict
aA	b	Shift
		shift-reduce conflict: $S \rightarrow aA$
aAb	ϵ	Reduce: $A \rightarrow b$
		reduce-reduce conflict: $C \rightarrow Ab$
aAA	ϵ	Reduce: $S \rightarrow aAA$
S	ϵ	Done

The sequence of operations shown is the one that leads to a successful parse, but alternative (unsuccessful) sequences are possible at all the points where a conflict is flagged.

To deal with these conflicts, we again make use of a stack or a queue to record alternative choice points. Below is the algorithm for shift-reduce parsing (to minimise clutter, we leave out the part that actually constructs the derivation tree):

Shift-Reduce Parsing

parse(p)

1. Reduced := ϵ; Unreduced := p;
   StackOrQueue := empty;
   ADD(⟨Reduced,Unreduced⟩, StackOrQueue);
2. REPEAT UNTIL either (i) Reduced = s and Unreduced = ϵ,
   or (ii) StackOrQueue is empty:
   (a) REMOVE(⟨Reduced,Unreduced⟩, StackOrQueue);
   (b) FOR EACH rule $A \rightarrow r_1 \ldots r_n$, such that Reduced = $ur_1 \ldots r_n$
   ADD(⟨uA, Unreduced⟩, StackOrQueue);
   (i.e. do all possible reductions)
   (c) IF Unreduced $\neq \epsilon$,
   i.e. Unreduced = av where a is a non-terminal
   THEN ADD(⟨Reduceda, v⟩, StackOrQueue);
   (i.e. shift)
3. IF Reduced = s and Unreduced = ϵ
   RETURN parse tree (constuction of which is not shown)
   ELSE FAIL

It is worth noting that depth-first bottom-up parsing does not run into the same the same termination problems with left recursion that top-down parsing does. (Why?)

7.4 LL(k) Parsing

In general, context free grammars are not amenable to deterministic top-down parsing. But a subclass of context free grammars, known as LL(k) grammars[3], can be deterministically parsed. This means that each point in the parse, it is possible to decide exactly which rule needs to be used to expand out the leftmost non-terminal symbol. This decision is made on the basis of looking ahead at the next k terminal symbols in the string to be parsed.

7.4.1 Lookahead

The following grammar illustrates how k-symbol lookahead can be used to select the correct rule for parsing. In this case, $k = 1$.

$$S \rightarrow aS \mid cA$$

[3]The "LL" stands for Left-to-right scanning of the input string to produce a Leftmost derivation.

$A \rightarrow bA \mid cB \mid \epsilon$
$B \rightarrow cB \mid a \mid \epsilon$

Top-down parsing with a one symbol lookahead is deterministic for the string *acbb*. Note that at each point in the parse, we will have generated a sentential form with a terminal prefix occurring before the leftmost non-terminal. Thus in a sentential form like acA, the terminal prefix is ac. By comparing this to the string to be parsed ($acbb$), we can see that the next terminal symbols to come are bb, and the first of these, b, constitutes the one word lookahead

	Derivation	Rule	Prefix Generated	Lookahead Symbol
S	$\Rightarrow aS$	$S \rightarrow aS$	ϵ	a
	$\Rightarrow acA$	$S \rightarrow cA$	a	c
	$\Rightarrow acbA$	$A \rightarrow bA$	ac	b
	$\Rightarrow acbbA$	$A \rightarrow bA$	acb	b
	$\Rightarrow acbb$	$A \rightarrow \epsilon$	acbb	ϵ

For example, at the first step of the derivation, the one symbol lookahead allows us to ignore the rule $S \rightarrow cA$ for expanding S. This is because we need to expand the leftmost non-terminal, S, to a string beginning with an a (the lookahead symbol). But the rule $S \rightarrow cA$ will only generate strings beginning with a c. Similarly at the second step, we can discount the rule $S \rightarrow aS$ since the next lookahead symbol is a c, but this rule cannot generate a string beginning with a c.

Other grammars may require looking ahead by more than one symbol to select the correct rule. Consider three alternative grammars that generate $a^{i+1}bc^i$: G1, G2 and G3.

G1: $S \rightarrow aaAc$
$A \rightarrow aAc$
$A \rightarrow b$

G2: $S \rightarrow aA$
$A \rightarrow Sc$
$A \rightarrow abc$

G3: $S \rightarrow aSc$
$S \rightarrow aabc$

In grammar G1, we need to use lookahead to see which of the two rules expanding A to use. The two rule both introduce terminal symbols on the extreme lefthand side of of the string the rewrite A to. Moreover, they are distinct terminal symbols, a and b. Hence for G1 it is only necessary to look ahead by one symbol to decide which rule to use: G1 is an LL(1) grammar.

In grammar G2 there are again two rules rewriting A. But the first of them, $A \rightarrow Sc$, does not directly introduce a leftmost terminal. However, if we look at the rule expanding $S \rightarrow aA$, we see that the S always expands out to have an initial a. If we look at ways of deriving strings from A we get

1. $A \overset{A \to Sc}{\Longrightarrow} Sc \overset{S \to aA}{\Longrightarrow} aAc \overset{A \to Sc}{\Longrightarrow} aScc \overset{S \to aA}{\Longrightarrow} aaAcc \Rightarrow \ldots$

2. $A \overset{A \to Sc}{\Longrightarrow} Sc \overset{S \to aA}{\Longrightarrow} aAc \overset{A \to abc}{\Longrightarrow} aabcc$

3. $A \overset{A \to abc}{\Longrightarrow} abc$

This shows that all terminal strings derivable from A begin with an a, whichever of the rules $A \to Sc$ and $A \to abc$ are used. However, all derivations that start by using the rule $A \to Sc$ (derivations 1 and 2) generate strings beginning with aa, whereas derivations using $A \to abc$ generate strings beginning with ab. Hence, by looking two symbols ahead we can determine which of the rules $A \to Sc$ and $A \to abc$ to use: G2 is an LL(2) grammar.

For grammar G3 we need to lookahead three symbols to decide which of the rules expanding S to choose. Again, if we look at possible derivations from S we get

1. $S \overset{S \to aSc}{\Longrightarrow} aSc \overset{S \to aSc}{\Longrightarrow} aaScc \overset{S \to aSc}{\Longrightarrow} aaaSccc \Longrightarrow \ldots$
2. $S \overset{S \to aSc}{\Longrightarrow} aSc \overset{S \to aSc}{\Longrightarrow} aaScc \overset{S \to aabc}{\Longrightarrow} aaaabccc$
3. $S \overset{S \to aSc}{\Longrightarrow} aSc \overset{S \to aabc}{\Longrightarrow} aaabcc$

4. $S \overset{S \to aabc}{\Longrightarrow} aabc$

We can see that all derivations that start by applying the rule $S \to aSc$ generate strings beginning with at least three as. However, derivations with the rule $S \to aabc$ produces strings beginning with aab. Hence looking ahead three symbols is enough to determine which rule to use: G3 is an LL(3) grammar.

The grammars shown so far are *strong LL(k) grammars*. This means that we *only* need to look at the next k terminal symbols to be derived, in order to select the correct rule for expanding the leftmost non-terminal. Below we will introduce a wider class of LL(k) grammars, but first we must be more formal about the notion of lookahead.

7.4.2 Lookahead Sets

Given some grammar, we can define the lookahead set $LA(N)$ for some non-terminal symbol N as follows

Lookahead set for non-terminal N: $LA(N)$
Let Grammar = $\langle NonTerms, Terms, Start, Rules \rangle$

$$LA(N) = \{x \mid Start \overset{*}{\Rightarrow} uNv \overset{*}{\Rightarrow} ux, \text{ where } ux \in Terms^*\}$$

In other words, take all the sentential forms of the grammar, uNv, where N is the leftmost non-terminal, and gather together all the terminal strings that Nv can be expanded out into.

We can give an analogous definition for the lookahead set associated with a particular grammar rule

Lookahead set for rule: $LA(N \rightarrow w)$
Let Grammar = $\langle NonTerms, Terms, Start, Rules \rangle$

$$LA(N \rightarrow w) = \{x \mid Start \stackrel{*}{\Rightarrow} uNv \stackrel{N \rightarrow w}{\Longrightarrow} uwv \stackrel{*}{\Rightarrow} ux, \text{ where } ux \in Terms^*\}$$

This is just a subset of the lookahead set for the symbol N, were the first rule used to generate terminal strings from Nv is the rule $N \rightarrow w$.

It is usual to *truncate* the lookahead sets associated with non-terminals and rules. That is, we only want to look at the first k symbols of any terminal string in the lookahead set. We define a truncation operation on sets of strings X, $trunc_k(X)$, as follows:

$$trunc_k(X) = \{u \mid u \in X \ \& \ length(u) \leq k, \text{ or } uv \in X \ \& \ length(u) = k\}$$

We then apply the truncation operation to the lookahead sets defined above to get k-length lookahead set. That is

$$LA_k(N) = tunc_k(LA(N))$$

$$LA_k(N \rightarrow w) = tunc_k(LA(N \rightarrow w))$$

7.4.3 Strong LL(k) Grammars and Parsers

We can now define what it is for a grammar to be stong LL(k). For each non-terminal symbol N, the k-length lookahead sets for the different rules expanding N must be disjoint. That is

Definition of Strong LL(k) Grammar

A grammar is strong LL(k) if for each non-terminal N:
for any pair of rules $N \rightarrow w_i$, $N \rightarrow w_j$:

$$LA_k(N \rightarrow w_i) \cap LA_k(N \rightarrow w_j) = \{\}$$

The idea behind this is that you only need to look at the next k symbols in the string to be parsed to uniquely select the correct rule to expand the leftmost non-terminal N. The top-down parsing alogorithm for strong LL(k) grammars is thus

Deterministic parser for strong LL(k) grammars:
parse(p)

1. SentForm := S;

2. REPEAT

   (a) Let Sentform == uLv, where L is the leftmost non-terminal;

   (b) Let p == ux;

   (c) k-Lookahead := $trunc_k(x)$;

   (d) Find rule $L \rightarrow w$ such that k-Lookahead $\in LA_k(L \rightarrow w)$;

   (e) SentForm := uwv

   UNTIL either
   (a) Sentform = p, or
   (b) k-Lookahead $\notin LA(L)$

3. IF Sentform = p SUCCEED
   ELSE FAIL

Some useful facts about strong LL(k) grammars are (see if you can work out why they are true):

1. If G is strong LL(k) for some k, G is unambiguous
2. If G has a left recursive non-terminal, then G is not strong LL(k) for any k.
3. All LL(1) grammars are strong LL(1)

As stated previously, not all LL(k) grammars are strong LL(k). We now turn to this wider class of grammar.

7.4.4 LL(k) Grammars and Parsers

Recall that a strong LL(k) grammar is one where the k-length lookahead sets for all the rules expanding any given non-terminal N are disjoint. With merely LL(k) grammars, the lookahead sets for some rules expanding the same non-terminal N may overlap. However, by looking also at the sentential form in which N occurs, it is still possible to select a single rule.

Example Here is an example of an LL(2) grammar that is not strong LL(2):

$$S \to Aabd \mid cAbcd$$
$$A \to a \mid b \mid \epsilon$$

We can construct the length 2 lookahead sets for the rules as follows

- $LA_2(S \to Aabd) = \{aa, ab\}$
  - $S \overset{S \to Aabd}{\Longrightarrow} Aabd \overset{A \to a}{\Longrightarrow} aabd$
  - $S \overset{S \to Aabd}{\Longrightarrow} Aabd \overset{A \to b}{\Longrightarrow} abbd$
  - $S \overset{S \to Aabd}{\Longrightarrow} Aabd \overset{A \to \epsilon}{\Longrightarrow} abd$
- $LA_2(S \to cAbcd) = \{ca, cb\}$
  - $S \overset{S \to cAbcd}{\Longrightarrow} cAbcd \overset{A \to a}{\Longrightarrow} cabcd$
  - $S \overset{S \to cAbcd}{\Longrightarrow} cAbcd \overset{A \to b}{\Longrightarrow} cbbcd$
  - $S \overset{S \to cAbcd}{\Longrightarrow} cAbcd \overset{A \to \epsilon}{\Longrightarrow} cbcd$
- $LA_2(A \to a) = \{aa, ab\}$
  - $Aabd \overset{A \to a}{\Longrightarrow} aabd$
  - $cAbcd \overset{A \to a}{\Longrightarrow} cabcd$
- $LA_2(A \to b) = \{ba, bb\}$
  - $Aabd \overset{A \to b}{\Longrightarrow} babd$
  - $cAbcd \overset{A \to b}{\Longrightarrow} cbbcd$
- $LA_2(A \to \epsilon) = \{ab, bc\}$
  - $Aabd \overset{A \to \epsilon}{\Longrightarrow} abd$
  - $cAbcd \overset{A \to \epsilon}{\Longrightarrow} cbcd$

Note that the lookahead sets for the rules $A \to a$ and $A \to \epsilon$ overlap: they both contain ab. Thus the grammar is not strong LL(2). [4]

However, if you look more closely at the way the lookahead sets for these two rules are put together, you will notice the following. The rule $A \to a$ only gives rise to the string ab when it is expanded in the context of the sentential form $cAbcd$. The rule $A \to \epsilon$ only gives rise to the string ab when expanded in the context of the different sentential form $Aabd$

Thus provided we also look at the sentential form within which we want to expand an A to produce an initial ab, we can still pick just one rule to do it.

[4] As it happens, the grammar *is* strong LL(3).

7.4.5 Local Lookahead and Parsing

For LL(k) grammars we need to define lookahead sets for rules that are relativised to the sentential forms in which the leftmost non-terminal is being expanded. We will call this relativised lookahead set the local lookahead set

Definition of Local Lookahead

Let $N \rightarrow w$ be a rule,
and $Start \stackrel{*}{\Rightarrow} uNv$ where N is the leftmost non-terminal

$$LA(N \rightarrow w,\ uNv) = \{x \mid uNv \stackrel{*}{\Rightarrow} ux,\ \text{where } ux \in Term^*\}$$

This definition should be compared with the earlier one for (global) lookahead for a strong LL(k) grammar:

$$LA(N \rightarrow w) = \{x \mid Start \stackrel{*}{\Rightarrow} uNv \stackrel{N \rightarrow w}{\Longrightarrow} uwv \stackrel{*}{\Rightarrow} ux,\ \text{where } ux \in Terms^*\}$$

The global lookahead for a rule is basically the union of all local lookaheads for different sentential forms uNv.

A grammar is LL(k) if the local lookaheads for all rules expanding any non-terminal are disjoint (note that as before, we truncate the lookahead sets to get k-length sets):

Definition of LL(k) Grammar

A grammar is LL(k) if for each non-terminal N:
For each sentential form $Start \stackrel{*}{\Rightarrow} uNv$:
For any pair of rules $N \rightarrow w_i$, $N \rightarrow w_j$:

$$LA_k(N \rightarrow w_i, uNv) \cap LA_k(N \rightarrow w_j, uNv) = \{\}$$

The algorithm for parsing with an LL(k) grammar is exactly like that for parsing with strong LL(k) grammars, expect that rules are chosen on the basis of both the lookahead string and the current sentential form:

Deterministic parser for LL(k) grammars:
parse(p)

1. SentForm := S;

2. REPEAT

   (a) Let Sentform == uLv, where L is the leftmost non-terminal;

   (b) Let p == ux;

   (c) k-Lookahead := $trunc_k(x)$;

   (d) Find rule $L \rightarrow w$ such that k-Lookahead $\in LA_k(L \rightarrow w, uLv)$;

   (e) SentForm := uwv

   UNTIL either
   (a) Sentform = p, or
   (b) k-Lookahead $\notin LA(L)$

3. IF Sentform = p SUCCEED
   ELSE FAIL

However, one major difference is that while it is feasible to compute in advance the global lookaheads for all rules in a strong LL(k) grammar, it is not feasible for local lookaheads in an LL(k) grammar. This is because most grammars will derive an infinite number of sentential forms uNv, so each rule will have an infinite number of local lookahead sets. Instead, the lookahead sets are computed during parsing. We now turn to an efficient way of doing this using $FIRST$ and $FOLLOW$ sets.

7.4.6 $FIRST$ and $FOLLOW$ Sets

We will first define truncated $FIRST$ and $FOLLOW$ sets

$\underline{FIRST_k(N)}$
set of k-length prefixes of terminal strings derivable from N

$$FIRST_k(N) = trunc_k(\{x \mid N \stackrel{*}{\Rightarrow} x, x \in Term^*\}$$

where N is a single non-terminal symbol

$\underline{FIRST_k(uv)}$
set of k-length prefixes of terminal strings derivable from uv

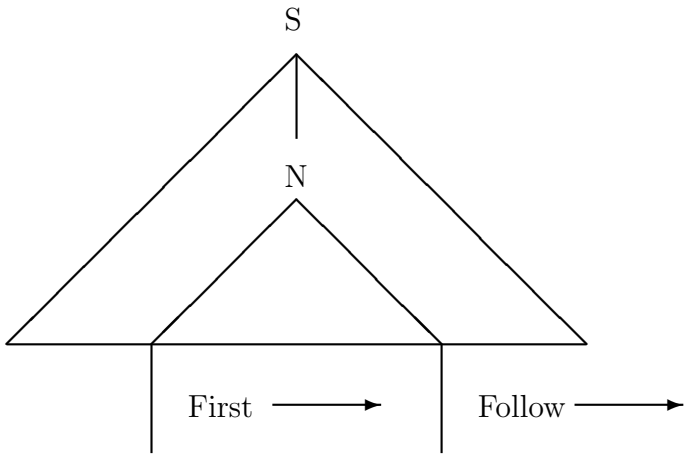
$$FIRST_k(uv) = trunc_k(\{x \mid uv \stackrel{*}{\Rightarrow} x, x \in Term^*\}$$

where uv is some string of terminal and/or non-terminal symbols

$\underline{FOLLOW_k(N)}$
set of k-length prefixes of terminal strings that can follow strings derived from N in sentential forms.

$$FOLLOW_k(N) = \{x \mid S \stackrel{*}{\Rightarrow} uNv \ \& \ x \in FIRST_k(v)\}$$

We can represent the FIRST and FOLLOW strings for a non-terminal N in an arbirary derivation as follows:



The FIRST string is the first k words derivable from N. The FOLLOW string is the first k words following the string derived from N. First and follow sets gather together these strings for all posssible derivations in the grammar.

Lookahead sets can be defined in terms of first and follow sets:

- $LA_k(N) = trunc_k(FIRST_k(N)FOLLOW_k(N))$
- $LA_k(N \to w) = trunc_k(FIRST_k(w)FOLLOW_k(N))$
- $LA_k(uNv) = FIRST_k(Nv)$
  where $S \stackrel{*}{\Rightarrow} uNv$ and $u \in Term^*$
- $LA_k(uNv, N \to w) = FIRST_k(wv)$

Having defined global (strong LL) and local (LL) lookahead sets in terms of truncated FIRST and FOLLOW sets, it remains to present algorithms for calculating the FIRST and FOLLOW sets. The algorithms are relatively straightforward, and the details do not need to be committed to memory

Algorithm for calculating $FIRST_k(N)$

1. FOR each $a \in Term$ DO $F'(a) = \{a\}$;
2. FOR each $N \in NonTerm$ DO
$$F(N) = \begin{cases} \{\epsilon\} & \text{if } N \to \epsilon \text{ is a rule} \\ \{N\} & \text{otherwise} \end{cases}$$
3. REPEAT
   (a) FOR each $N \in NT$ DO $F'(N) = F(N)$;
   (b) FOR each rule $N \to u_1 u2 \ldots u_n$ with $n > 0$ DO
   $F(N) = F(N) \cup trunc_k(F'(u_1)F'(u_2)\ldots F'(u_n))$;

   UNTIL $F(N) = F'(N)$ for all $N \in NT$
4. $FIRST_k(N) = F(A)$

Algorithm for calculating $FOLLOW_k(N)$

Assume that we have the $FIRST_k(N)$ sets for every $N \in NonTerm$

1. INIT: $FL(S) = \{\epsilon\}$; FOR each $N \in (NT - \{S\})$, DO $FL(N) = \{\}$;
2. REPEAT
   (a) FOR each $N \in NT$, DO $FL'(N) = FL(N)$
   (b) FOR each rule $N \rightarrow w$, where $w = u_1 u_2 \ldots u_n \notin V^*$, DO
   - $L = FL'(N)$;
   - IF $u_n \in NT$, THEN $FL(u_n) = FL(u_n) \cup L$;
   - FOR $i = n - 1$ TO 1 DO
     $L = trunc_k(FIRST_k(u_{i+1})L)$;
     IF $U_i \in NT$ THEN $FL(u_i) = FL(u_i) \cup L$;

   UNTIL $FL(N) = FL'(N)$ for every $N \in NT$
3. $FOLLOW_k(N) = FL(N)$

7.5 LR(k) Parsing

We now turn to deterministic bottom-up parsing, or LR(k) parsing. Here, the "LR" stands for "Left to right scanning of the input string to produce a Rightmost derivation". Once again, k stands for the number of words we lookahead in the input string, but we will mainly focus on deterministic parsing that uses no lookahead.

Recall that to make top-down parsing deterministic, we need some sort of oracle to select the correct rule to expand each leftmost non-terminal. With bottom-up parsing we need an oracle that says whether to (a) shift or reduce, and (b) if reduce, with which rule. The oracle thus resolves any shift-reduce or reduce-reduce conflicts. For LR(0) grammars, the oracle is able to give its answer without using lookahead.

In this section, we will first introduce the idea of LR(0)-contexts and viable prefixes. We will then show how a finite state automaton can provide the oracle required for deterministic shift reduce parsing. And finally, we will say a few words about LR(k) parsing where $k > 0$.

7.5.1 LR(0) Contexts and Viable Prefixes

LR(0) Contexts

Suppose that a grammar permits a rightmost derivation

$$S \stackrel{*}{\Rightarrow} uRv \Rightarrow uwv \stackrel{*}{\Rightarrow} xv$$

(where $xv \in Term^*$). That is, there is at least one way in which a sentence can be derived by applying the rule $R \to w$ to the sentential form uRv. Corresponding to this component of a rightmost derivation, there will also be a reduction step in a shift-reduce parse:

Reduced	Unreduced	Operation
ϵ	xv	...
uw	v	Reduce: $R \to w$
uR	v	...
...		
S	ϵ	

What this means is that there is at least one parse where the rule $R \to w$ can be applied to reduce the string uw, and which leads finally to a reduction to the start symbol. (Not all parses applying $R \to w$ to uw are guaranteed to succeed: it all depends on what words are remaining in v).

uw is an LR(0)-Context for $R \to w$

iff there is a rightmost derivation

$$S \overset{*}{\Rightarrow} uRv \Rightarrow uwv \overset{*}{\Rightarrow} xv$$

Where $uw \in (Term \cup NonTerm)^*$, $xv \in Term^*$.

This means that in a bottom-up shift-reduce parse, if you encounter uw as the reduced string, then it is sensible to attempt a reduction with the rule $R \to w$.

Suppose, on the other hand, that uw is not an LR context for $R \to w$. This means that there is no successful rightmost derivation that applies the rule $R \to w$ to a string uRv. Thus, there is no point in attempting to reduce uw with $R \to w$ in a shift-reduce parse. Although the rule matches locally, permitting a reduction to uR, there will still be no possible derivation in the grammar including such a step.

Suppose that that we knew the contexts for all the rules in the grammar. Then at any point in a shift-reduce parse where a reduction is possible (i.e. we have a string uw where w matches the RHS of some rule), we can eliminate those rules with which it is pointless to attempt a reduction: if uw is not a context for $N \to w$, even though the rule matches, do not reduce with it.

Moreover, if the LR contexts for all the rules are disjoint (any string uw is a context for at most one rule), then whenever a reduction is possible, there is only one rule that it is sensible to reduce with. That is, we can use the (disjoint) contexts to resolve reduce-reduce conflicts.

Viable Prefixes

Suppose that u is the prefix of some LR(0) context uw. Then we say that u is a *viable prefix* of the grammar

Viable Prefix

If uw is an LR(0) context for a rule $R \rightarrow w$,
then any prefix (initial substring) of uw is a viable prefix of $R \rightarrow w$

If, while doing a shift-reduce parse, we encounter a reduced string that is not a context for any rule, but is the viable prefix for one (or more) rules, then it is sensible to shift. After this, depending on what words are shifted on, it may be possible to do a reduction. (Again, there is no guarantee of this: it depends on what words get shifted onto the end of the reduced string).

It is possible, with some grammars, for a context of one rule to be a viable prefix for another. In such cases we are left in a shift-reduce quandary: either reduce with the rule whose context it is, or shift in the hope of an eventual reduction with the rule whose prefix it is. But if no context for any rule is the prefix for any other, then all shift-reduce conflicts can be resolved.

7.5.2 Definition of LR(0) Grammar

We have already hinted at the definition of an LR(0) grammar in describing contexts and viable prefixes: disjoint contexts resolve reduce-reduce conflicts, and no contexts serving as prefixes resolve shift-reduce conflicts.

Definition of LR(0) Grammar
A grammar is LR(0) iff

(a) Contexts for all rules are disjoint

Hence: for a given reduced string/context, at most one sensible rule to reduce with.

(b) No context for one rule is a viable prefix for another rule

Hence: for a given reduced string, no choice between reducing now or shifting some more to reduce later

Note: not all context free grammars are LR(0).

Although we have defined what LR(0) contexts and viable prefixes are, we have said nothing about how they can be recognized. We now turn to this.

7.5.3 LR(0) Machines

An LR(0) machine is a (deterministic) finite state automaton that accepts strings of terminal and non-terminal symbols that are either LR(0) contexts or viable prefixes. Moreover, for those strings that are LR(0) contexts, it says which rule (or rules, if the grammar is not LR(0)) the string is a context for.

The LR(0) machine is used as an oracle in deterministic, LR(0), shift-reduce parsing. At each point in th parse, the current reduced string is fed as input

to the LR(0) machine. The machine then decides to shift or reduce, and which rule to reduce with. In particular, the LR(0) machine can:

(a) Terminate, identifying a context for a (single) rule
REDUCE

(b) Terminate, identifying a viable prefix for some rule or rules
SHIFT

(c) Not terminate successfully
FAIL Parse

LR(0) machines can be constructed for any context free grammar, even if the grammar is not LR(0). Moreover, inspection of the LR(0) machine can be used to determine whether or not the grammar is LR(0). The construction first produces a non-deterministic machine from the rules of the grammar, and then applies the standard technique to determinise the machine.

Non-Deterministic LR(0) Machines: Dotted Rules

The states of the non-deterministic LR(0) machine are provided by dotted rules (or LR(0)-items) taken from the grammar rules. These are obtained by putting a single dot at all possible positions on the right hand side of a rule. So, for example, a rle with three symbols on its RHS gives rise to four dotted rules:

$N \to aBc$ gives four dotted rules:

$$\{N \to .aBc,\ N \to a.Bc,\ N \to aB.c,\ N \to aBc.\}$$

More formally:

1. If $A \to uv$ is a rule, then $A \to u.v$ is an LR(0) item
2. If $A \to \epsilon$ is a rule, then $A \to .$ is an LR(0) item

Note that an item $A \to u.$ (with the dot at the end) is called a complete item.

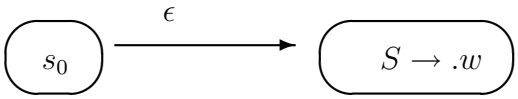
Each item / dotted rule serves as an (accepting) state in the non-deterministic LR(0) machine. In addition to these, there is a single extra start state, s_0. The transitions between states are defined as follows

Transition Function for Non-Deterministic LR(0) Machine

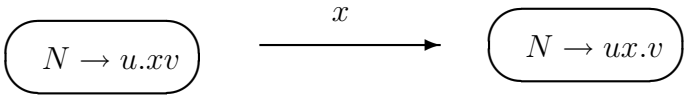
1. $\delta(q_0, \epsilon) = \{S \to .w \mid S \to w \in R\}$
2. $\delta(A \to u.av, a) = \{A \to ua.v\},\ a \in Term$
3. $\delta(A \to u.Bv, B) = \{A \to uB.v\},\ A \in NonTerm$
4. $\delta(A \to u.Bv, \epsilon) = \{B \to .w \mid B \to w \in R\}$

In other words

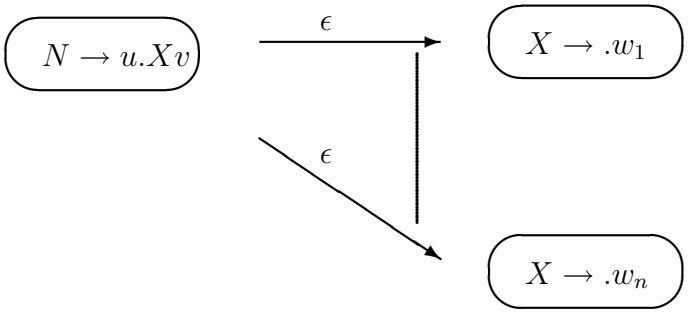
1. There is an empty transition from the start state s_0 to every state $S \rightarrow .w$ corresponding to a rule expanding the start symbol S with the dot at the beginning.



2. For any state $N \rightarrow u.xv$ where x is a terminal or non-terminal symbol, there is an x-transition to the state that moves the dot one place further along:



3. For any state $N \rightarrow u.Xv$ where X is a non-terminal symbol, add further empty transitions to all the states expanding X where the dot is at the beginning, $X \rightarrow .w$:



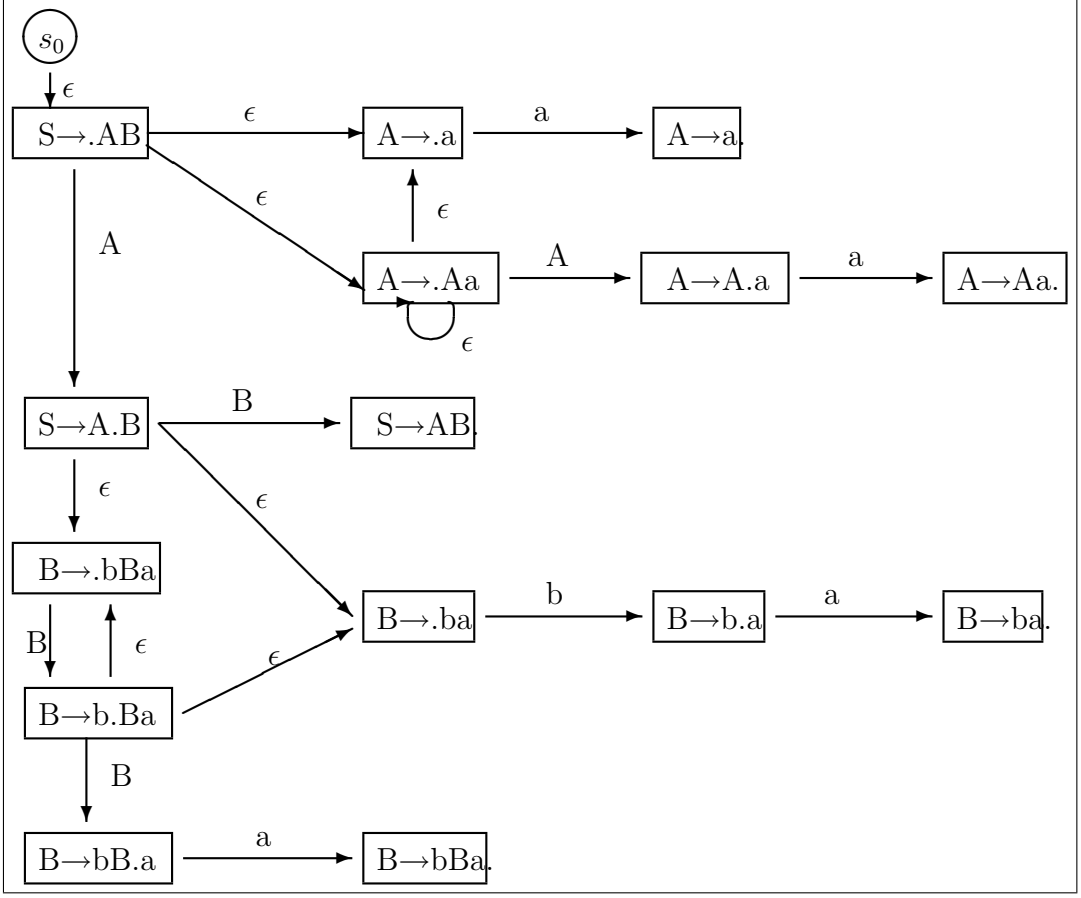
Example To illustrate the construction of a non-deterministic LR(0) machine, we will construct machines for the grammar:

$$S \rightarrow AB$$
$$A \rightarrow Aa \mid a$$
$$B \rightarrow bBa \mid ba$$

(which happens to be LR(0)).

Non-Deterministic LR(0) Machine

Rule	LR(0)-Items
$S \to AB$	$\{S \to .AB,\ S \to A.B,\ S \to AB.\}$
$A \to Aa$	$\{A \to .Aa,\ A \to A.a,\ A \to Aa.\}$
$A \to a$	$\{A \to .a,\ A \to a.\}$
$B \to bBa$	$\{B \to .bBa,\ B \to b.Ba,\ B \to bB.a,\ B \to bBa.\}$
$B \to ba$	$\{B \to .ba,\ B \to b.a,\ B \to ba.\}$

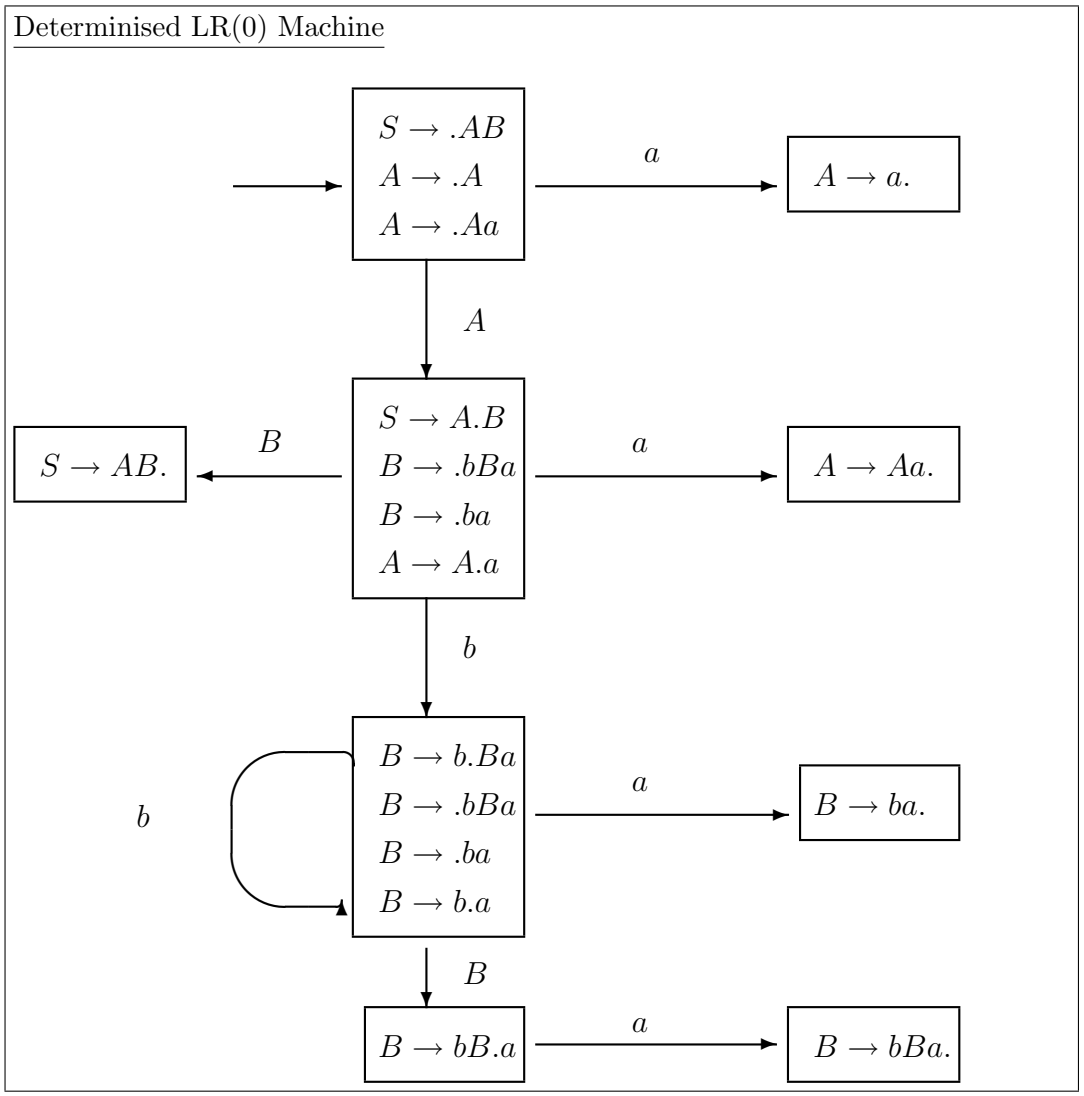


Note that in both the machines drawn, *all* states are accepting states, though we have neglected to draw double boxes around them.

Determinising LR(0) Machines

Using the standard technique for determinising finite state machines, the non-deterministic LR(0) machine can be determinised. This yields an automaton defining exactly the same language as before. What is this language? The machine accepts all strings that are either LR contexts or viable prefixes.

Taking the example above, we can determinise the machine to get the following



The machine can be run on any sequence of terminal and/or non-terminal symbols from the grammar.

(a) If the machine successfully terminates in a state containing a complete item (i.e. a rule with a dot at the end), then the sequence of terminals and/or non-terminals is an LR(0) context for the rule.

Hence, reduction with the rule is a possibility.

For example, running the machine on Aba terminates in state $\{B \to ba.\}$, indicating a reduction via the rule to AB.

(b) If the machine successfully terminates in a state containing a rule where the dot is not at the end, then the sequence of terminals and/or non-terminals is a viable prefix for the rule.

Hence, further shifting may lead one to the context for a rule.

For example, running the machine on Ab terminates in state $\{B \to$

$b.Ba, B \rightarrow .bBa, B \rightarrow .ba, B \rightarrow b.a\}$ indicating that Ab is a prefix for the rules $B \rightarrow bBa$ and $B \rightarrow ba$, and that a shift should take place.

(c) If the machine does not successfully terminate (i.e. gets stuck in a state with remaining input but no possible transitions), then the sequence of symbols is neither a context nor a viable prefix.

For example, running the machine on b gets stuck in the initial state.

Looking at the machine above, we can see that the grammar is LR(0). The disjointness of contexts corresponds to the fact that there are no states in the determinised machine containing more than one complete item. That no context for one rule is a viable prefix for another corresponds to the fact that there are no states mixing complete and incomplete items. In other words, we can tell the grammar is LR(0) by the fact that all states containing complete items contain only a single complete item.

If a state contained two complete items, this would indicate that the contexts for those two rules overlapped. If a state contained a complete item plus some incomplete items, this would indicate that the context for one rule is a prefix for another. Note however, it is permissible for a state to contain several different incomplete items: a string can be a viable prefix for several different rules, even in an LR(0) grammar

LR(0) Machines for Non-LR(0) Grammars

It is important to note that LR(0) machines can be constructed for any context free grammar, whether or not the grammar is LR(0). Indeed, the construction of the determinised LR(0) machine can be used to see whether or not the grammar is LR(0).

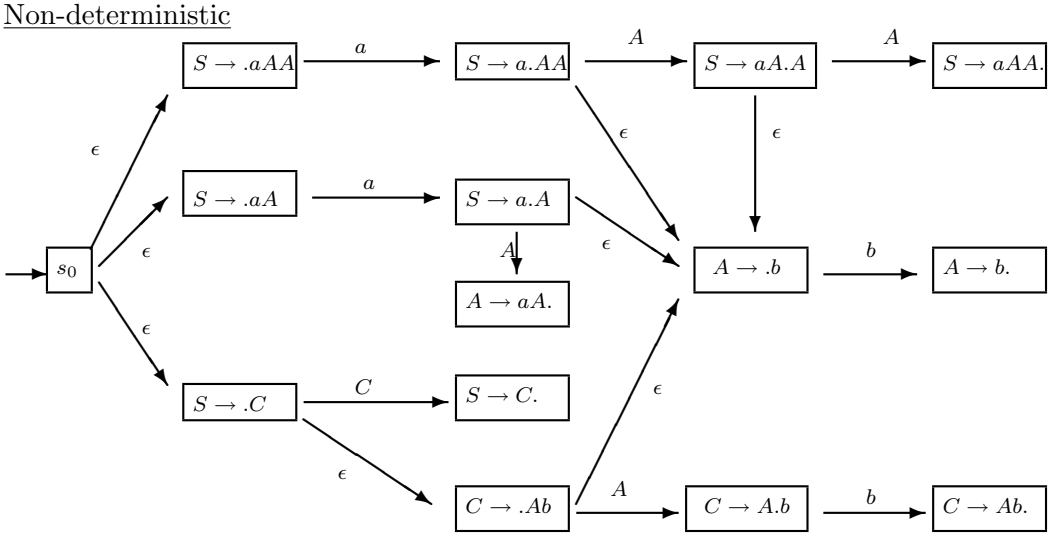
As an example, here is a non-LR(0) grammar with its dotted rules shown:

Rule	LR(0)-Items
$S \rightarrow aA$	$\{S \rightarrow .aA,\ S \rightarrow a.A,\ S \rightarrow aA.\}$
$S \rightarrow aAA$	$\{S \rightarrow .aAA,\ S \rightarrow a.AA,\ S \rightarrow aA.A.\ S \rightarrow aAA.\}$
$S \rightarrow C$	$\{S \rightarrow .C,\ S \rightarrow C.\}$
$A \rightarrow b$	$\{A \rightarrow .b,\ S \rightarrow b.\}$
$C \rightarrow Ab$	$\{C \rightarrow .Ab,\ C \rightarrow A.b,\ C \rightarrow Ab.\}$

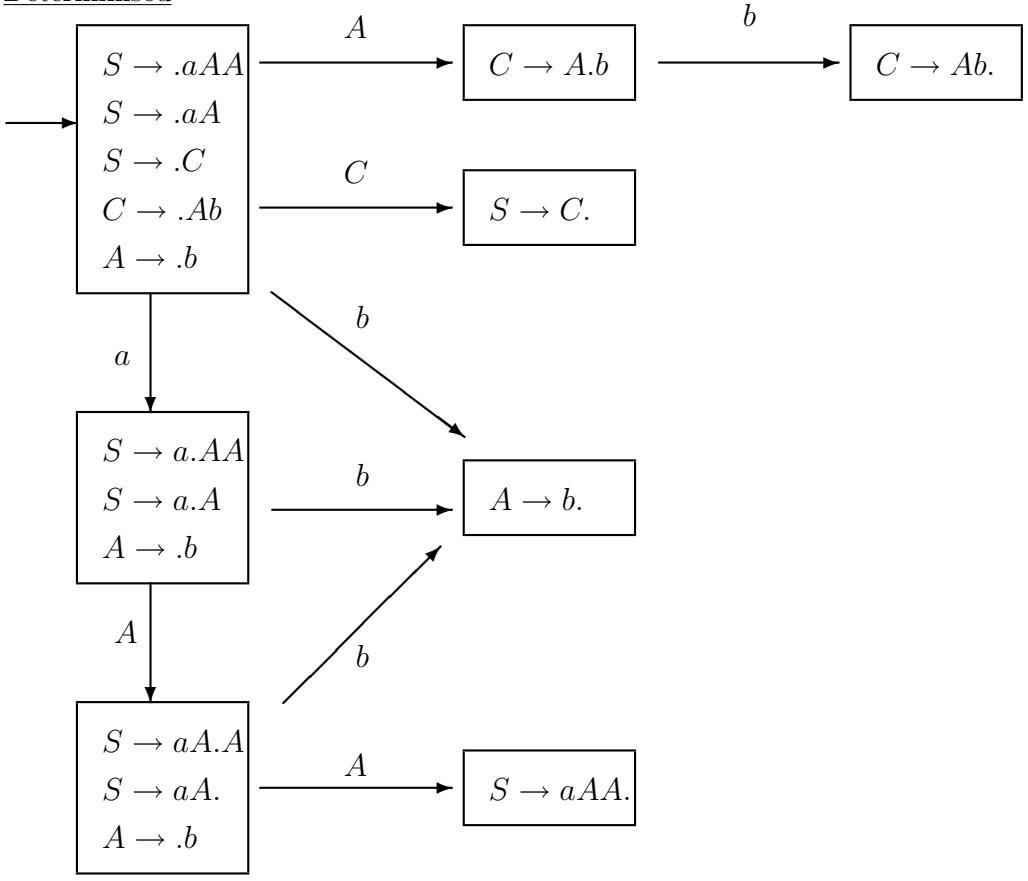
And overleaf are the non-deterministic and non-deterministic LR(0) machines:

LR(0) Machines for non-LR(0) Grammar

Non-deterministic



Determinised



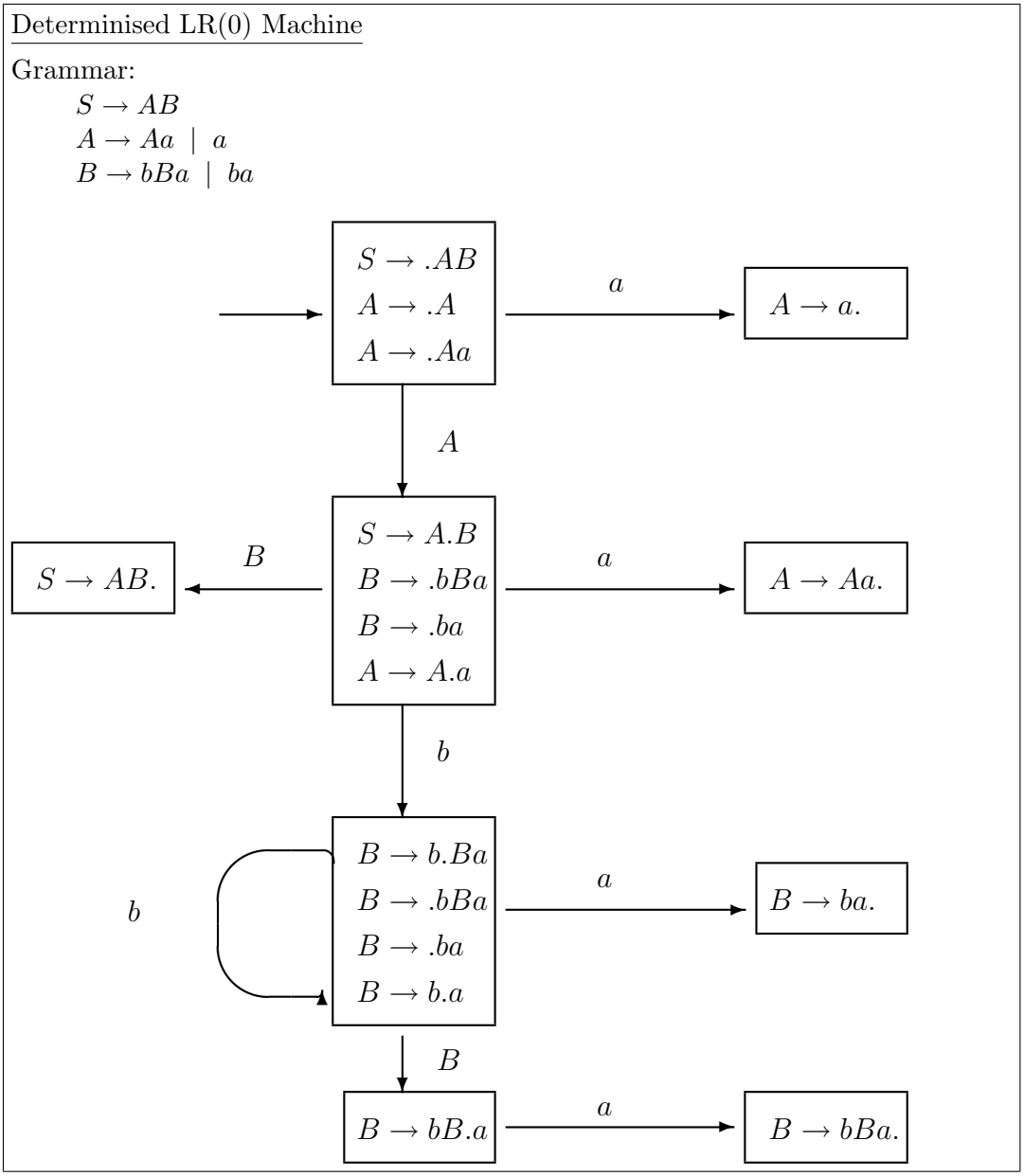
Note the state in the determinised machine labelled

$$\{S \to aA.A,\ S \to aA.,\ A \to .b\}$$

This mixes a complete item, $S \to aA.$ with two incomplete items. A string like aA, which terminates in this state is thus both a context for the rule $S \to aA$, and also a prefix for the rules $S \to aAA$ and $A \to b$. This shows that the grammar is not LR(0).

7.5.4 Deterministic Parsing with LR(0) Machines

Let us go back to a grammar that *is* LR(0), and its determinised LR(0) machine.



We can use the machine as an oracle for a shift-reduce parser. The algorithm for a deterministic LR(0) parser is as follows, where δ^* is the extended transition

function of the grammar's deterministic LR(0) machine, and s_0 its start state:

LR0-parse(p)

1. $Reduced := \epsilon$; $Unreduced := p$
2. REPEAT UNTIL Reduced = S:
   (a) IF $\delta^*(s_0, Reduced) = \{A \to w.\}$, THEN reduce with rule $A \to w$
   (b) ELSEIF $\delta^*(s_0, Reduced)$ contains $A \to y.z$, THEN shift
   (c) ELSE $\delta^*(s_0, Reduced)$ is undefined, THEN FAIL

We can illustrate this algorithm by doing a deterministic shift reduce parse of the string *aabbaa* with the grammar and machine shown

	Reduced	Unreduced	FSA	Result	Action
1	ϵ	*aabbaa*	$\delta^*(s_0, \epsilon) =$	$\{S \to .AB, A \to .a, A \to .Aa\}$	shift
2	a	*abbaa*	$\delta^*(s_0, a) =$	$\{A \to a.\}$	reduce
3	A	*abbaa*	$\delta^*(s_0, A) =$	$\{A \to A.a, S \to A.B,$ $B \to .bBa, B \to .ba\}$	shift
4	Aa	*bbaa*	$\delta^*(s_0, Aa) =$	$\{A \to Aa.\}$	reduce
5	A	*bbaa*	$\delta^*(s_0, A) =$	$\{A \to A.a, S \to A.B,$ $B \to .bBa, B \to .ba\}$	shift
6	Ab	*baa*	$\delta^*(s_0, Ab) =$	$\{B \to .bBa, B \to b.Ba,$ $B \to .ba, B \to b.a\}$	shift
7	Abb	*aa*	$\delta^*(s_0, Abb) =$	$\{B \to .bBa, B \to b.Ba,$ $B \to .ba, B \to b.a\}$	shift
8	$Abba$	a	$\delta^*(s_0, Abba) =$	$\{B \to ba.\}$	reduce
9	AbB	a	$\delta^*(s_0, AbB) =$	$\{B \to bB.a\}$	shift
10	$AbBa$	ϵ	$\delta^*(s_0, AbBa) =$	$\{B \to bBa.\}$	reduce
11	AB	ϵ	$\delta^*(s_0, AB) =$	$\{S \to AB.\}$	reduce
12	S	ϵ			

Note, for example, how at step 4 a reduce-reduce conflict between the rules $A \to Aa$ and $A \to a$ is resolved.

7.5.5 LR(k) ($k \geq 1$) Grammars

LR(0) grammars are in general to restrictive to define programming languages. However, most programming languages are deliberately designed to be LR(1). These can be deterministically parsed provided that the shift-reduce oracle has access not only to the reduced part of the string, but also to the next word (or k words) remaining in the input. It is possible to produce LR(1) machines, somewhat like LR(0) machines, that recognize LR(1)-contexts and viable prefixes.

We will not describe the construction of LR(1) machines here (see Sudkamp, Chap. 16 for more information). Instead, we will just define what an LR(1) context is

LR(1) Context

A string uwz is an LR(1) context for a rule $R \rightarrow w$ iff there is a rightmost derivation

$$S \stackrel{*}{\Rightarrow} uRv \Rightarrow uwv \stackrel{*}{\Rightarrow} xv$$

where $xv \in Term^*$ and z is the first symbol of v, or ϵ if $v = \epsilon$

In other words, if you have a reduced string of uw in a shift-reduce parser, and the next symbol to shift is z, then it is sensible to attempt a reduction with the rule $R \rightarrow w$.

A grammar is LR(1) when (a) the LR(1) contexts for all rules are disjoint, and (b) no context for one rule is the prefix of another context.

The following grammar is not LR(0), but is LR(1):

$S \rightarrow A \mid Bc$
$A \rightarrow aA \mid a$
$B \rightarrow a \mid ab$

Consider four different derivations

Reduced	Unreduced	Operation
ϵ	abc	Shift
a	bc	Shift
ab	c	Reduce
B	c	Shift
Bc	ϵ	Reduce
S		

Reduced	Unreduced	Operation
ϵ	ac	Shift
a	c	Reduce
B	c	Shift
Bc	ϵ	Reduce
S		

Reduced	Unreduced	Operation
ϵ	aaa	Shift
a	aa	Shift
aa	a	Shift
aaa	ϵ	Reduce
aaA	ϵ	Reduce
aA	ϵ	Reduce
A	ϵ	Reduce
S		

Reduced	Unreduced	Operation
ϵ	a	Shift
a	ϵ	Reduce
A	ϵ	Reduce
S		

Note that when the parser has the reduced string a, there are three possible actions

1. Reduce with $A \rightarrow a$
2. Reduce with $B \rightarrow a$

3. Shift to obtain either aA or ab

The second stage in all the parses above involves scanning a reduced string a and choosing the appropriate operation. Which of the above three operations is appropriate can be determined by looking at the next symbol in the unreduced string:

Reduced	Next Word	Operation
a	ϵ	Reduce with $A \rightarrow a$
a	a	Shift
a	b	Shift
a	c	Reduce with $B \rightarrow a$

This is the essence of LR(1) parsing.

Unix provides a utility, `yacc` (standing for "yet another compiler compiler") that will automatically construct an LR(1) machine for a grammar, and report on remaining possibiilties for shift-reduce and reduce-reduce conflicts if the grammar is not LR(1).

7.6 Summary

This has been a lengthy chapter, but only begins to cover some of the aspects concerned with parsing context free grammars. We have covered

- Top-Down vs Bottom-Up (shift-reduce) parsing
- Depth-first vs Breadth-first search
- Deterministic, LL(k), top-down parsing
- Deterministic, LR(k), bottom-up parsing

Having absorbed the material of this chapter, you should ensure that you are able to (i) perform depth-first or breadth-first top-down parses, (ii) perform shift-reduce parses and recognize potential shift-reduce and reduce-conflicts, (iii) describe what makes a grammar LL(k), and (iv) describe what makes a grammar LR(0) and how to detect this condition.

Chapter 8

Pumping Lemma & Closure Properties

In this chapter we look at the pumping lemma for context free languages, and then at a few facts about how context free languages can (and cannot) be combined to form other context free languages

8.1 The Pumping Lemma

The pumping lemma for context free languages is similar to that for regular languages. It capitalises on the fact that automata for CFLs, push-down stack automata, have only a restricted stack-like memory. There are certain things a single stack cannot do, and hence certain languages that cannot be context free.

In particular, all context free languages are infinite (since all finite languages are regular). This infiniteness is obtained (in the grammar) by means of recursive rules, e.g.

$$S \rightarrow aSb \mid ab$$

Limitations on the stack like memory of PDAs mean that if a recursion can occur once, then it can occur any number of times. That is, context free languages cannot place limits on the number of times a recursion can occur in a derivation.

8.1.1 Statement of Pumping Lemma

The pumping lemma for CFLs is formally stated as follows

Pumping Lemma for Context Free Languages

If L is a CFL, then there is a number k such that any string $z \in L$ where $length(z) > k$ can be written as $z = uvwxy$ where

1. $length(vwx) \leq k$
2. $length(v) + length(x) > 0$, and
3. $uv^n wx^n y \in L$ for any $n \geq 0$

As with the pumping lemma for regular languages, this is more often written in the negative form, and used to show that a given language is *not* context free:

Negated CFL Pumping Lemma

L is not a CFL if there is a string $z \in L$ where $length(z) > k$,
such that for all possible decompositions of $z = uvwxy$, where

1. $length(vwx) \leq k$
2. $length(v) + length(x) > 0$

there is some number n such that

$$uv^n wx^n y \notin L$$

8.1.2 Applying the Pumping Lemma

We will illustrate the use of the CFL pumping lemma to establish that the language $a^n b^n c^n$ $(n \geq 0)$ is not context free.

First we have to pick on a string of length greater than k which cannot be successfully decomposed. As with the pumping lemma for regular languages, we do not actually know what the value of k is. So we need to pick some string guaranteed to be longer than k.

Let us pick the string

$$z = a^k b^k c^k$$

which clearly has a length greater than k, whatever the value of k is.

We now need to consider all possible ways of decomposing the string z such that

1. $z = uvwxy$
2. $length(vwx) \leq k$
3. $length(v) + length(x) > 0$

Here is a table showing all possible decompositions

u	v	w	x	y
a^p	a^q	a^r	a^s	$a^{(k-p-q-r-s)}b^kc^k$
a^p	a^q	a^r	$a^{(k-p-q-r)}b^s$	$b^{(k-s)}c^k$
a^p	a^q	$a^{(k-p-q)}b^r$	b^s	$b^{(k-r-s)}c^k$
a^p	$a^{(k-p)}b^q$	b^r	b^s	$b^{(k-q-r-s)}c^k$
a^kb^p	b^q	b^r	b^s	$b^{(k-p-q-r-s)}c^k$
a^kb^p	b^q	b^r	$b^{(k-p-q-r)}c^s$	$c^{(k-s)}$
a^kb^p	b^q	$b^{(k-p-q)}c^r$	c^s	$c^{(k-r-s)}$
a^kb^p	$b^{(k-p)}c^q$	c^r	c^s	$c^{(k-q-r-s)}$
$a^kb^kc^p$	c^q	c^r	c^s	$c^{(k-p-q-r-s)}$

For each possible decomposition we kind find a value of n such the uv^nwx^ny is not a string of the form $a^ib^ic^i$:

u	v	w	x	y	n
a^p	a^q	a^r	a^s	$a^{(k-p-q-r-s)}b^kc^k$	$n = 0$
a^p	a^q	a^r	$a^{(k-p-q-r)}b^s$	$b^{(k-s)}c^k$	$n = 2$
a^p	a^q	$a^{(k-p-q)}b^r$	b^s	$b^{(k-r-s)}c^k$	$n = 0$
a^p	$a^{(k-p)}b^q$	b^r	b^s	$b^{(k-q-r-s)}c^k$	$n = 2$
a^kb^p	b^q	b^r	b^s	$b^{(k-p-q-r-s)}c^k$	$n = 0$
a^kb^p	b^q	b^r	$b^{(k-p-q-r)}c^s$	$c^{(k-s)}$	$n = 2$
a^kb^p	b^q	$b^{(k-p-q)}c^r$	c^s	$c^{(k-r-s)}$	$n = 0$
a^kb^p	$b^{(k-p)}c^q$	c^r	c^s	$c^{(k-q-r-s)}$	$n = 2$
$a^kb^kc^p$	c^q	c^r	c^s	$c^{(k-p-q-r-s)}$	$n = 0$

For example, if either v or x contain a mixture of different letters, then repeating them twice will lead to a string where the as, bs and/or cs are scrambled up, e.g.

$$a^pa^{2q}a^ra^{(k-p-q-r)}b^sa^{(k-p-q-r)}b^sb^{(k-s)}c^k$$

And when the v and x do not contain mixtures of letters, the repeating them zero times will leave too many occurrences of either as, bs and/or cs, e.g.

$$a^pa^{(k-p-q)}b^rb^{(k-r-s)}c^k \;=\; a^{(k-p)}b^{(k-s)}c^k$$

We have thus shown that for each possible decomposition of the string $a^kb^kc^k$ meeting the required length constraints, we cannot freely repeat the v and x parts of the decompositions.[1] This establishes that the language $a^nb^nc^n$ is not context free.

8.1.3 Proving the Pumping Lemma

In the section we show why the context free pumping lemma works. The lemma relies on the fact that strings above a certain length will inevitably contain

[1]For this case we do not need to labouriously write out every decomposition. We can take shortcuts by noting that (a) any decomposition that mixes the letters in v and/or x cannot be freely repeated, and (b) after a while there is a symmetry to the remaining decompositions.

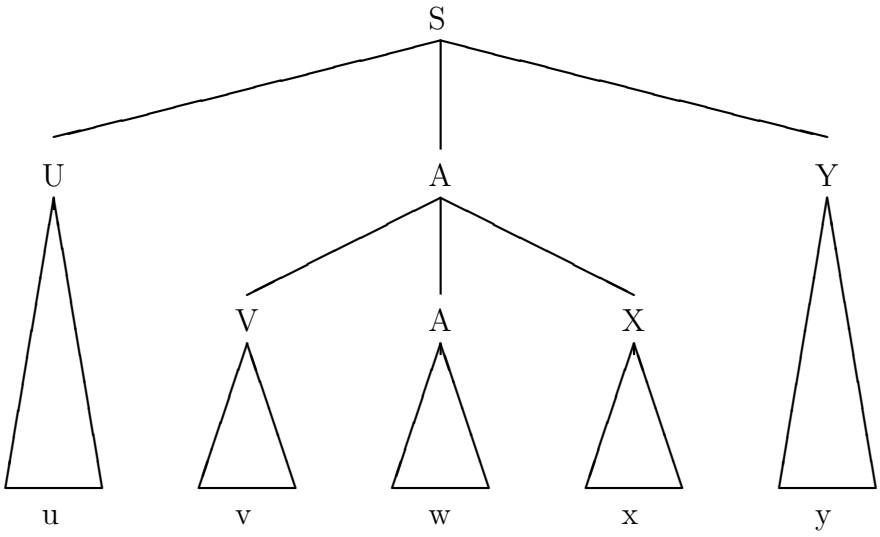
recursion in their CFG derivations. To prove this, first assume a grammar for the language in Chomsky Normal Form. This means that all derivations will give rise to binary trees.

For binary trees of depth n[2] it is easy to show that the terminal string given by the leaf nodes has a length less than or equal to 2^{n-1}. Conversely, if the string has length $\geq 2^n$, the binary tree has a depth of at least $n+1$

Now suppose we have a grammar in CNF containing only n non-terminal symbols. This means that if a derivation tree contains a path of length $\geq n+1$, then the derivation must contain a recursion. This is because at least one of the non-terminals must occur more than once on the path of length $\geq n+1$ from root to leaf. In other words, any string of length 2^n or greater must involve a recursion in its derivation from a grammar in Chomsky Normal Form.

So we now arrive at the first part of the pumping lemma: any string of length $\geq k$, (where $k = 2^n$, and n is the number of non-terminals in the CNF grammar), must involve at least one recursion in its CNF derivation.

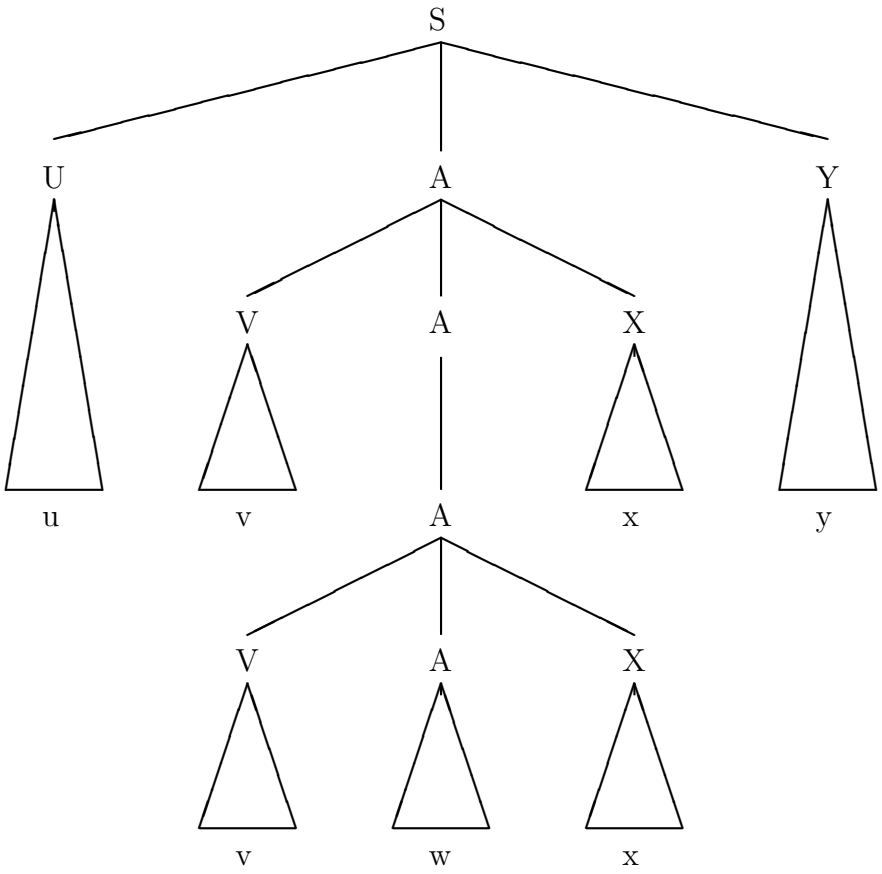
That is, the derivation tree must take the general form, with at least one recursion on some symbol A



Here, the string w is what is produced by the non-recursive derivation from the lower A. The sub-strings v abd x flank this, and are derived in the course of the recursion on the upper A (via a rule like $A \rightarrow VAX$).

There is no reason why the recursion on A should not be repeated a second time, to derive a different string, shown below:

[2]The depth of a tree is the maximum number of nodes you have to pass through in following a path from the root node to a leaf node.



Here there are two copies if v and x flanking the w terminating the A-recursion. And we could repeat the recursion again, giving three copies of v and x flanking w and so on. And of course, we could also leave out the recursion altogether, so that there are no copies of v and x flanking w.

We can thus see that any string of length $> k$ must contain at least recursion, and so is decomposible into a string $uvwxy$. Moreover, the v and the x must be freely repeatable, so that derivations for $uv^n wx^n y$ for any $n \geq 0$ should also be possible. Moreover, assuming that the sub-derivation $A \stackrel{*}{\Rightarrow} vwx$ does not contain any other recursions than the single one on A, it follows that $length(vwx) < k$.

8.2 Closure Properties of Context Free Languages

Here we state a few facts about context free languages

1. Context free languages are closed under union, concatenation and Kleene star

- Union:
  1. Rename non-terminals so that they are distinct in the two grammars.
  2. Introduce a new start symbol, S, and two new rules $S \rightarrow S_1$ and $S \rightarrow S_2$ where S_1 and S_2 are the start symbols of the two component grammars.
  3. The new grammar if CF, and will generate strings in either grammar
- Concatenation:
  1. Rename non-terminals so that they are distinct in the two grammars.
  2. Introduce a new start symbol, S, and a new rule $S \rightarrow S_1 S_2$
  3. The new grammar is CF, and will generate strings of the first grammar concatenated with that of the second
- Kleene star:
  1. Add a new start symbol, and the rules $S \rightarrow S_1 S$ and $S \rightarrow \epsilon$

2. CFLs are not closed under intersection or complementation
   - Counter example for intersection:
     Let $L_1 = \{a^i b^i c^j\}$, $L_2 = \{a^j b^i c^i\}$
     The intersection of these is $\{a^i b^i c^i\}$, which is not regular
   - Counter example for complementation:
     If CFLs are closed in complementation, then $L = \overline{\overline{L_1} \cup \overline{L_2}}$ should be CF if L_1 and L_2 are.
     But $\overline{\overline{L_1} \cup \overline{L_2}} = L_1 \cap L_2$, and we know CFLs are not closed under intersection.
3. The intersection of a CFL with a regular language is a CFL
   (Proof: see Sudkamp 8.5.3)

8.3 Summary

The facts about closure of CFLs may occasionally be useful. You need to be able to apply the pumping lemma for CFLs (and for regular languages), but you do not really need to be able to reconstruct the proofs of the lemmas.

Part III

Turing Machines

Chapter 9

Turing Machines

We now reach the top of the Chomsky Hierarchy. Turing machines, and the language(s) associated with them are not of major practical use (unlike regular and context free languages). However, they do allow us to establish some surprising results about the limits of what can be computed. Turing machines are equivalent in power to digital computers (with infinite memory); and it is possible to prove that there are some things that Turing machines (and hence digital computers) just cannot do.

In this chapter we introduce Turing machines, which are essentially finite state automata with unlimited random access memory. In the next chapter we briefly discuss the language(s) defined by Turing machines. And in the final chapter, we use Turing machines to establish some bounds on what can be computed.

9.1 The Standard Turing Machine

A finite state automaton is a finite state machine (a newtwork of states and transitions) without any memory. Transitions are determined solely on the basis of the current state and the next word on the input string.

Pushdown automata are finite state machines with a stack memory. Transitions are determined by the current state, the next word on the input string and the symbol popped off the top of the stack. Making the transition pushes a new symbol onto the top of the stack.

Turing machines are finite state machines with infinite random access memory. The memory is represented as an infinitely long tape divided into squares, where each square can hold at most one symbol. Typically, the input string is written onto the tape. Transitions depend on the current state and the symbol written in the tape location currently being scanned. Taking a transition, as well as moving to another state, can rewrite the symbol being scanned on the tape and/or move one square to the left or the right on the tape.

It is customary to talk of there being a *tape head* that at any time scans a single square on the tape. Transitions are thus dependent on

(a) the current state

(b) the symbol currently being scanned by the tape head

The effects of taking a transition are to

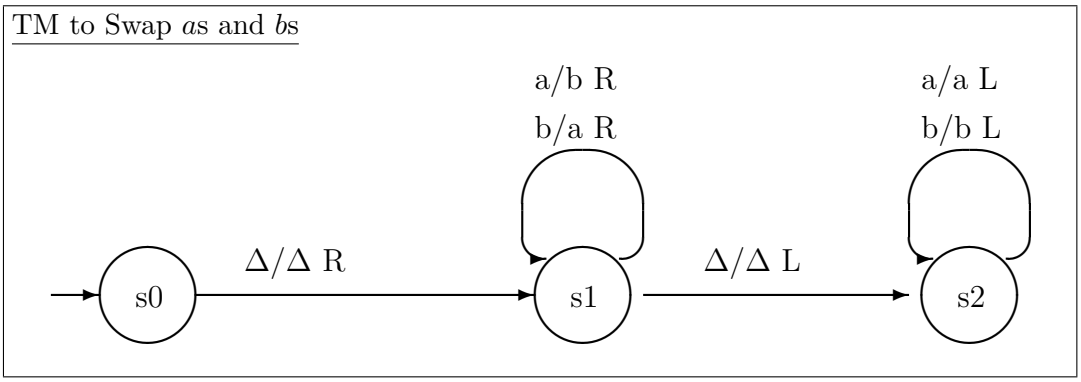
(a) move to a (possibly) new state

(b) rewrite the symbol in the tape position currently being scanned

(c) move the tape head either one square to the left or one square to the right

In a standard Turing machine, the squares on the tape are numbered from position 0 up to infinity (the tape is infinite towards the right, but has a beginning — position 0 — on the left).

Example 1 Here is the graphical representation for a simple Turing machine:



The diagram is a state-transition network, with heavily labelled transitions. The the transition label "$a/b\ R$" means

1. Take the transition if a is currently lying under the tape head
2. Rewrite the tape with a b in that position
3. Move the tape head one place to the right

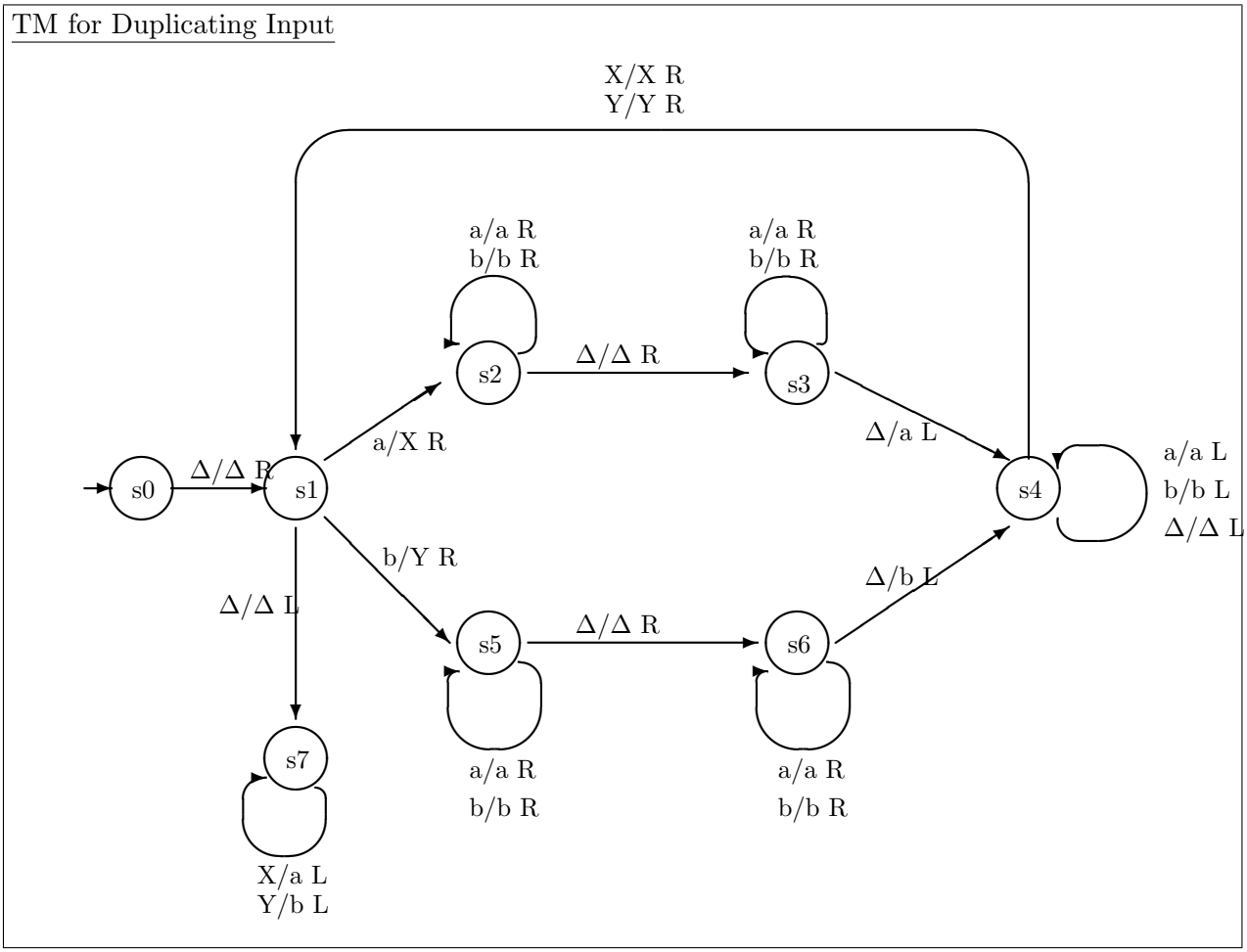
The transition "$\Delta/\Delta\ L$" means take the transition if a blank (Δ) is lying under the tape head, rewrite it with a blank, and move one square to the left.

It is customary to write the input string for a Turing machine on a blank tape, where the input string starts at position 1. The machine starts off in the start state (s_0) scanning position 0 of the tape. The machine halts when it can take no further transitions.

We start the machine shown by writing a string of as and bs on an otherwise blank tape, starting at position 1. The machine itself begins in state s_0 with the tape head scanning position 0.

The transition out of state s_0, Δ/Δ R, is triggered by reading the blank in tape position 0, leaves the symbol as it is, and moves the tape head one square to the right. There are two transitions looping around on state $s1$. One of them is triggered by reading an a under the tape head, rewrites it to a b, and moves the tape head one square to the write. It thus swaps as by bs. The other loop is similar, except that it swaps bs by as. The machine will continue looping around on state s_1, moving the tape head one square to the right and swappping as and bs, until it reaches the end of the input string. The end of the input is signalled by the first blank on the tape, and at this point a transition is made to state s_2, whereupon the tape head is moved back one square to the left. There are two looping transitions on state s_2, triggered by reading as or bs, leaving the symbols as they are, and moving the tape head back one square to the left. The loops continue until the tape head moves back to the blank in position 0 of the tape. At this point, no further transitions are possible. So the Turing machine halts, with a sequence of as and bs on an otherwise blank tape, but where the as and bs have been swapped around compared to the input string.

Example 2 Here is a somewhat more complex Turing machine that will duplicate a string of sumbols written on a tape. That is, you start with $\Delta u \Delta \ldots$ written on the tape, and end with $\Delta u \Delta u \Delta \ldots$.



Transitions out of s_1 replace the first a with a X or the first b with a Y. The top and bottom halves of the machine then read through the rest of the string, and whatever has so far been copied, until the tape head reaches the second blank terminating the copied string.

Different branches are followed depending on whether an a or a b is being copied. The second blank is then replaced by an a or a b as appopriate (transitions out of s_3 and s_6 respectively). State s_4 then loops round taking the tape head back along the tape until it reaches an X or Y.

At this point, it moves the tape head forward, and goes back to s_1 so that the next character in the input string can be copied. If there are no more characters to be copied (i.e. a blank follows the X or Y), then s_7 loops back through the tape replacing Xs by a's and Ys by b's. The machine halts in s_7 when the tape head reaches the blank in position 0.

TMs can do Anything (Almost)!

The Turing machine to duplicate a string is a relatively complex combination of very simple components (and in case you are worrying: you do not need

to remember how this particular TM, or any other, works). According to the *Church-Turing Thesis*, which will be discussed later, Turing machines can be constructed to do anything that a digital computer can — in fact, anything that can be algorithmically computed can be computed using a Turing machine.

This is of theoretical rather than practical significance: Turing machines do not provide a high-level programming language. If anything, they are considerably worse than machine or assembler code. But the fact that TMs can be constructed from such simple building blocks does allow a number of interesting theoretical results to be established.

9.1.1 Formal Definition of Standard TMs

We now briefly give the formal definition of a standard Turing machine

A standard TM has a tape that begins in position 0, and which extends indefinitely far to the right. The tape positions are numbered, 0, 1, 2, ...

The machine is 5-tuple

$$M = \langle S, \Gamma, V, \delta, s_0 \rangle$$

where
S is a finite set of states
Γ is the tape vocabulary, including the special symbol Δ to represent a blank
V is the input vocabulary, and $V \subseteq \Gamma - \{\Delta\}$
δ is a partial transition function from $S \times \Gamma$ to $S \times \Gamma \times \{L, R\}$
s_0 is the initial state

In a given state, the transition function consults the symbol currently lying under the tape head. It then moves to a (possibly) new state, rewrites the symbol on the tape, and moves the tape head either to the left or the right, depending on whether the third item in the function's result is L or R.

If, for a given state and tape symbol, no transitions are defined, then the machine halts. If the machine attempts to move the tape head back past position 0, the machine crashes.

To recognise a string, the tape is initially all blank, and the string is written on to it starting at position 1. If the machine halts, the string is recognised

9.2 Variants on Turing Machines

There are a number of variants on the standard Turing machine, some of them looking very different, but which all turn out to be equivalent. In this chapter we will describe some of the variants and sketch the reasons for why they are equivalent to standard TMs

9.2.1 TMs Accepting by Final State

The standard TM has no final states. A string u is accepted if

1. the string is written on an otherwise blank tape, starting at position 1
2. the machine starts in its initial state, scanning tape position 0
3. the machine halts in some state

We can also specify final states for the TM, so that (3) above becomes

3'. the machine halts in a final state

To convert a standard TM to one that accepts by halting in a final state: make every state a final state.

To convert a TM that accepts by halting in a final state to a standard TM:

(a) Add a new looping state

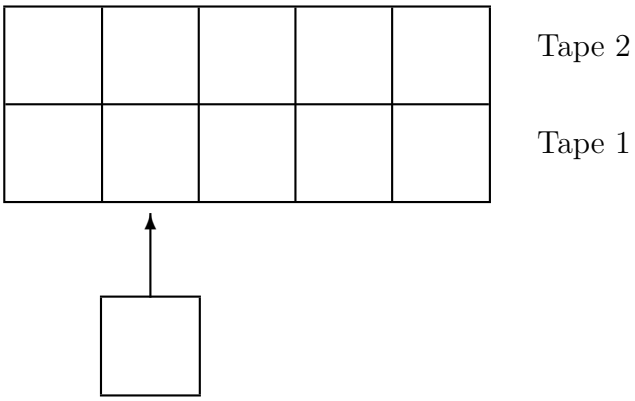
This has a transition that loops back onto the same state and moves the tape head to the right, whatever symbol is currently being scanned.

Whenever you get into a looping state, the TM will continue round, moving the tape head inifintely far to the right.

(b) For every non-final state, and for every tape symbol for which no transition is defined for that state, (i.e. every failing configuration) add a transition to the looping state

9.2.2 Multi-Track TMs

A multitrack TM is one where several tapes are glued together side by side:



That is, the tape head can read or write several symbols at a time, one from each tape.

Given that the tape vocabulary is finite, we can create new complex symbols. These will be pairs (n-tuples) of symbols from the vocabulary of all the tapes. Thus if tape 1 has the symbol a, and tape 2 has the symbol A, we create a new symbol $\langle a, A\rangle$. Reading the symbol $\langle a, A\rangle$ is equivalent to reading the symbol a and the symbol A off two tapes.

9.2.3 Two-Way Tapes

These are TMs where the tape extends indefinitely in both directions (there is no start of the tape).

We can simulate a one-way tape on a two-way machine by putting a start of tape marker just to the left of the initial tape head position. That is, the first action of the TM will be to move the tape head to the left, write the start of tape marker, and move right. If during computation the tape head reads the start of tape marker, the transition function moves to a non-accepting state that terminates the computation.

To simulate a two-way tape with a one way tape, imagine that the two-way tape has been folded in half, to give a two-track tape. That is, the tape head reads or writes two symbols at a time, one from the right hand half of the folded tape, and one from the left hand half. Each state s_i in the two-way TM gives rise to two states in one-way TM: $\langle s_i, U\rangle$ and $\langle s_i, D\rangle$ The U and D specify whether the transition should depend on the symbol on the upper tape or the lower tape. (The upper tape is the original left hand side of the tape).

The transitions on the U and the D states are the same as the transitions on the original s_i states except that

1. For U (D) state, we only pay pay attention to the symbol on the upper (lower) tape
2. For U states, when the original transition moved the tape head left (right), it now moves it right (left).
3. Upper states *normally* only make transitions to uppers states, and lower states to lower states

We also need to make sure that the machine writes a middle of tape marker at the start of the upper tape. Suppose we are in an upper state $\langle s_i, U\rangle$ scanning the middle of tape marker, with the symbol x on the lower tape. If in the original machine there was a transition from s_i on reading x that moved to state s_j and moved the tape head *right*
Then move to $\langle s_j, D\rangle$ and move the tape head right. If in the original machine there was a transition from s_i on reading x that moved to state s_j and moved the tape head *left*
Then move to $\langle s_j, U\rangle$ and move the tape head right.

9.2.4 Multi-Tape TMs

This is where there are several independent tape heads on several tapes. Transitions can move each of the tape heads in different ways. The input is put on tape 1, and all the others start blank.

To emulate a multitape TM on a multitrack TM: Suppose we have two independent tapes. We can emulate this with a 5-track TM. Two tracks will contain the contents of the two tapes Two tracks will be blank apart from a single symbol on each indicating the position of the tape head on the corresponding tape. The fifth track contains a single marker at the start. This is used to reposition the tape head to the start, so that it can be moved forwards to either of the two marked tape head positions on the other tracks.

9.2.5 Non-Deterministic TMs

A non-deterministic TM is one where more than one transition may be available for a given state / tape symbol pair. A non-deterministic (single tape) TM can be emulated on a deterministic three tape machine.

We assume that all the transitions for the NTM have been numbered, and that n is the maximum number of transitions for a state / tape symbol pair. For each numbered transition in the NTM, the number assigned gives the third tape symbol required to trigger a transition on the 3-tape DTM.

A computation on DTM proceeds as follows

1. Some sequence of number from 1 to n is written to tape 3
2. The input string on tape 1 is copied to the standard position on tape 2
3. The computation on the input is simulated on tape 2, with the transition being determined by the number on tape 3
4. If the simulation halts, the machine halts and the input is accepted
5. A new sequence of numbers is generated on tape 3 and steps 2–5 are repeated.

In other words, a Turing machine has sufficient power to encode its own stack-ilke backtracking mechanism. Hence a deterministic TM can emulate the non-deterministic execution of a non-deterministic TM. Hence, like finite state automata, but unlike pushdown stack automata, non-determinism does not affect the expressive power of TMs.

9.2.6 Post Machines

A post machine has a queue (known as a store) of unlimited length. The machine consists of READ, ADD, ACCEPT and REJECT states and a single START state.

- The READ states read (and remove) the leftmost symbol on the store, and make a transition to a new state depending on the value read.
- The ADD states add a new symbol to the end (right) of the store, and make a transition to some other state (independent of the store contents or the symbol added.
- The ACCEPT and REJECT states have no transitions coming out of them, and do what their names suggest.
- The start state has no transitions coming into it, and does what its name suggests.

The store alphabet contains a special symbol, #, normally used as an end-of-input marker. The input to the machine is written to the store, with a # at the end. (Note that any ADDs will add symbols after the #)

Simulating a Post Machine on a TM

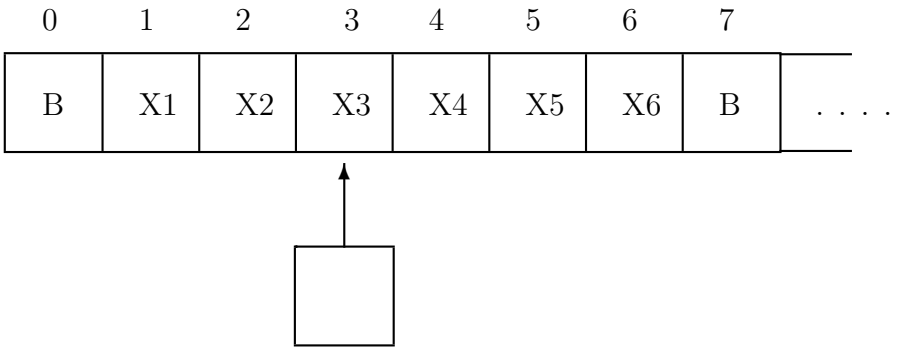
All we really have to do here is simulate a queue on the TM's tape.

Assume that the store contents occur in a contiguous block somewhere on the tape surrounded by blanks. READing a symbol amounts to moving to the start of the tape, moving right until a non-blank is found, replacing a symbol with a blank, and moving one place further right. ADDing a symbol involve moving to the end of the block of non-blanks, and rewriting the first non-blank as the symbol.

Simulating a TM on a Post Machine

To simulate a TM, we need to be able emulate a TM tape on a Post machine store. That is: we need to be able to read/write a symbol from/to the middle of the store.

Use the symbol # to mark the position of the TM tape head in the store. Characters to the left of the TM tape head are placed to the right of the #, and vice versa. Thus, the tape configuration



Corresponds to the store

$$X_4X_5X_6\#X_1X_2$$

To move the tape head right, we just read and remove the X_4 from the front of the store, and add a new symbol to the end of the store:

$$X_5X_6\#X_1X_2Y$$

Moving the tape head left is a little more involved. We need to read a symbol off the front of the queue, add the new symbol *to the front*, and then shift the last symbol on the store to the front:

$$X_2YX_5X_6\#X_1$$

We can get a PM to add a character to the front of a store, and read a character from the end of the store as follows. To add a to the front of the store

1. Add a special symbol, e.g. \$ to the end of the store, and then add a to the end of the store
2. Repeatedly read characters from the front ot the store and add them to the end, until you reach \$ (left shift).
3. Remove the \$ by reading it.

To read from the end of the store

1. Add \$ to the end of the store.
2. Read two characters, c_1 and c_2 from the front of the store
3. REPEAT UNTIL $c_2 = \$$

   IF $c_2 \neq \$$,
   THEN add c_1 to the end of the store
   read the next character c from the front,
   let $c_1 = c_2$, $c_2 = c$

   ELSEIF $c_2 = \$$,
   THEN add c_1 to the front of the store.
4. The last character on the store is now at the front, with the rest of the store unchanged, and we can read it normally

9.2.7 Two Stack PDAs

It turns out the a PDA with two stacks is equivalent to a TM. (Adding more stacks does not increase the power of the 2PDA).

9.3 Summary

This chapter has introduced the idea of a standard TM, and some variants that turn out to be expressively equivalent. You do not need to remember in detail why the variants are equivalent, but you should remember the fact that they are.

Chapter 10

Context Sensitive & Recursively Enumerable Languages

Having introduced Turing Machines, we now turn to the languages defined by them. In fact, there is a family of languages, depending on certain extra restrictions placed on the TM.

10.1 Recursive and Recursively Enumerable Languages

A standard TM accepts a string if (a) when the string is written in the standard position (starting at position 1) on an otherwise blank tape, then (b) the TM eventually halts. What, if anything, is written on the tape when the TM halts is immaterial.

However, this characterisation neglects the fact that some TMs for some inputs can spin off into a loop. TMs thus define two classes of language

Recursive Language

A TM defines a recursive language iff for every possible string it either

- halts (i.e. accepts the string), or
- crashes (i.e. rejects the string).

Recursively Enumerable Language

A TM defines a recursively enumerable language iff for every possible string it either

- halts (i.e. accepts the string), or
- crashes (i.e. rejects the string),
- loops forever (i.e. rejects the string)

Note that TMs defining recursive languages meet all the requirements for defining a recursively enumerable language. So the recursive languages are a subset of the wider class of recursively enumerable languages.

10.1.1 Unrestricted Rewrite Grammars

Recursively enumerable languages can also be defined by unrestricted rewrite grammars

Unrestricted Rewrite Grammar

An unrestricted rewrite grammar is a 4-tuple

$$G = \langle N, T, S, R \rangle$$

where
N is a set of non-terminal symbols
T is a set of terminal symbols
$S \in N$ is the start symbol
R is a set of rules of the form

$$(N \cup T)^* \rightarrow (N \cup T)^*$$

In an unrestricted rewrite grammar, any string of terminal and/or non-terminal symbols can be rewritten as any other string of terminals and non-terminals. The following are examples of unrestricted rewrite rules

$AB \rightarrow BA$
$aB \rightarrow BAC$

$a \rightarrow AB$
$ab \rightarrow ba$

Unrestricted rewrite grammars are of little practical interest.

10.1.2 TMs $\Leftrightarrow$ Unrestricted Rewrite Grammar

Here we briefly describe how TMs can be converted to unrestricted rewrite grammars and vice versa. This is for information only, and it is not necessary to remember the details of this.

Grammars to TMs

We can construct a 3-tape TM from an unrestricted rewrite grammar as follows.
Tape 1 holds the input.
Tape 2 holds the rules, where a rule $A \rightarrow w$ is represented on the tape as a string $A\#w$, and the rules are separated by #'s.
The derivations of the grammar are simulated on tape 3.

The computation on the machine proceeds as follows

1. S is written in position 1 on tape 3
2. The rules of G are written on tape 2
3. The input is written on tape 3.
4. A rule $A\#w$ is chosen from tape 2.
5. An instance of the string A is chosen on tape 3. If none exists, the computation halts in a rejecting state (and possibly backtracks)
6. The string A is replaced by w on tape 3
7. If the strings on tapes 1 and 3 match, computation halts in an accepting state
8. To try another rule, steps 3–7 are repeated

TMs to Grammars

Likewise, from a non-deterministic TM (with final states) we can construct an unrestricted rewrite grammar.

First, we need to represent a TM configuration as a string of symbols. If we are in state s_n with the tape head scanning the jth tape position, we represent this as the string

$$a_1 \dots a_{j_1} s_n a_j a_{j+1} \dots a_m \Delta$$

where $a_1, \dots, a_m$ are the symbols on the tape.

A transition, $\delta(s_i, x) = \langle s_j, y, R\rangle$ on a configuration us_ixv can be represented as a string transformation

$$us_ixv \Longrightarrow uys_jv$$

We use grammar rules to generate the string transformations corresponding to the effects of transitions in the TM. The derivation of a terminal string u in the grammar consists of three distinct subderivations (using three subsets of rules):

1. Generate from S the string $u[s_0\Delta u]$

   This sets up the TM simulation. It says we are in the start state with the tape head in position 0 and the input u in standard position — $[s_0\Delta u]$. The preceding u says that we are trying to generate a u

2. Simulation of the TM compututions on the string $[s_0\Delta u]$.

   This applies string transformations corresponding to TM transitions to the configuration recorded between square brackets

3. If the configuration is of the form $[vs_ixw]$, where s_i is an accepting state, and no transitions are defined for $\delta(s_i, x)$, then the simulation has halted in an accepting state.

   In this case we can remove all of the string between square brackets, just to leave the terminal string u

There is a standard trick for generating languages of the form $u[pu]$, where u is some terminal string and p a fixed terminal prefix. For each terminal symbol a_i, introduce a non-terminal A_i and rules

$$T[\rightarrow a_iT[A_i$$
$$S \rightarrow a_iT[a_i]$$

For each pair of terminals / non-terminals there is a swap rule

$$A_ia_j \rightarrow a_jA_i$$

For each terminal there is also a termination rule

$$A_i] \rightarrow a_i]$$

To introduce the prefix p (in this case $s_0\Delta$) we have rules

$$S \rightarrow [s_0\Delta]$$
$$T[\rightarrow [s_0\Delta$$

For $a_1 = a$ and $a_2 = b$, consider the following derivation of $aab[s_0\Delta aab]$

$$\begin{aligned} S &\rightarrow aT[a] \\ &\rightarrow aaT[Aa] \\ &\rightarrow aaT[aA] \\ &\rightarrow aaT[aa] \\ &\rightarrow aabT[Baa] \\ &\rightarrow aabT[aBa] \\ &\rightarrow aabT[aaB] \\ &\rightarrow aabT[aab] \\ &\rightarrow aab[s_0 \Delta aaB] \end{aligned}$$

The second set of rules, for simulating the TM computation on the string between the square brackets is derived directly from the TM transitions

- $s_i xy \rightarrow z s_j y$ whenever $delta(s_i, x) = \langle s_j, z, R \rangle$
- $s_i x] \rightarrow z s_j \Delta$ whenever $delta(s_i, x) = \langle s_j, z, R \rangle$
- $y s_i x \rightarrow s_j yz$ whenever $delta(s_i, x) = \langle s_j, z, L \rangle$

The third set of rules, for eliminating the string between square brackets if it represents an accepting configuration, is

- $s_i x \rightarrow E_R$ whenever $\delta(s_i, x)$ is undefined and s_i is an accepting state (i.e. halted in an accepting state)
- $E_R x \rightarrow E_R$
- $E_R] \rightarrow E_L$
- $x E_L \rightarrow E_L$
- $[E_L \rightarrow \epsilon$

When an accepting configuration is encountered, the rules first move the the right deleting symbols until they reach], and then move back to the left deleting symbols until they reach [.

10.2 Context Sensitive Languages & Linear Bounded Automata

We turn briefly to a class of languages midway between context free and recursive. These are the context sensitive languages, of which the language $a^i b^i c^i$ is a standard example, and they are defined by a restricted form of TM known as a linear bounded automaton.

10.2.1 Context Sensitive Grammars

A context sensitive grammar is an unrestricted rewrite grammar subject to one restriction: The righthand side of the rule cannot be shorter than the left hand side.

Unrestricted Rewrite Grammar

An unrestricted rewrite grammar is a 4-tuple

$$G = \langle N, T, S, R \rangle$$

where
N is a set of non-terminal symbols
T is a set of terminal symbols
$S \in N$ is the start symbol
R is a set of rules of the form

$$LHS \rightarrow RHS$$

where $LHS, RHS \in (N \cup T)^*$
and $length(LHS) \leq length(RHS)$

Thus a rule like $aBc \rightarrow ac$ could not be part of a context sensitive grammar.

The restriction on context sensitive rules means that in derivations the length of the derived string either remains constant or increases; the string can never go shorter.

CS Grammar for $a^i b^i c^i$

Recall that the pumping lemma for CFLs shows that $a^i b^i c^i$ is not a context free language. It is however context sensitive (provided $i > 0$)

$S \rightarrow aAbc \mid abc$
$A \rightarrow aAbC \mid abC$
$Cb \rightarrow bC$
$Cc \rightarrow cc$

Derivation

S
$a\underline{A}bc$
$aa\underline{A}bCbc$
$aaab\underline{Cb}Cbc$
$aaabbC\underline{Cb}c$
$aaabb\underline{Cb}Cc$
$aaabbbC\underline{Cc}$
$aaabbb\underline{Cc}c$
$aaabbbccc$

10.2.2 Linear Bounded Automata

Context sensitive languages are accepted by a restricted from of Turing machine known as linear bounded automata. This is a Turing machine where the lenth of the tape is limited to the length of the input plus 2. In other words, a finite state machine with strictly limited random access memory.

For proofs of the equivalence between context sensitive grammar and linear bounded automata, see Sudkamp or Cohen.

10.3 The Chomsky Hierarchy

This is a convenient place to review the Chomsky hierarchy to summarise the languages and machines we have encountered in this course:

Language	Grammar	Acceptor	Notes
Regular	LHS single non-terminal RHS either (a) a single terminal, or (b) a single terminal plus single non-terminal	Finite state machines	Regular expressions
Context Free	LHS single non-terminal RHS can be anything	(Non-Deterministic) Pushdown stack automata	Deterministic subset
Context Sensitive	RHS no shorter than LHS	Linear bounded automata	
Recursively Enumerable	No restrictions	Turing machines	TMs may reject by looping

The hierarchy represents an inclusion ordering over languages:

Inclusion Relations between Classes of Language

regular $\subset$
  deterministic context free $\subset$
    context free $\subset$
      context sensitive $\cup\{\{\epsilon\}\} \subset$
        recursive $\subset$
          recursively enumerable

This also shows a few of the intermediate languages we have encountered.

Recall that any language is a set of strings. A language class is a set of languages, i.e. a set of sets of strings. Thus the inclusion relation over language classes says that a a grammar or automaton higher up the Chomsky hierarchy can be used to define a wider set of languages than one lower down.

Chapter 11

Decidability & Computability

We finally turn to some theoretical results on the limits of computation that can be established via Turing machines. We start off with the Church-Turing thesis. This is a claim (unprovable, but well confirmed) that any algorithmically computable problem can be computed by a Turing machine. Or put another way, anything that a digital computer can do, a Turing Machine can do.

We then go on to discuss the halting problem. We have already pointed out that some TMs loop forever on certain inputs. The halting problem asks whether it is possible to design a TM which, when given the specification of another TM and its input, will determine whether or not that TM will halt. The answer turns out to be that it is not possible to design such a Turing machine. Hence, there is at least one well-defined and interesting problem that cannot be solved by a Turing machine. We then go on to show, from the unsolvability of the halting problem, that there are a range of other problems that are also unsolvable (or undecidable, as we shall call it). This includes a number questions about properties of context free grammars.

We finally briefly discuss computability in terms of Turing machines.

11.1 Church-Turing Thesis

The Church-Turing thesis essentially claims that any algorithmic procedure can be implemented on a Turing machine. This is not a claim that can mathematically be proved. One of the reasons for this is that we do not really have a well-defined, prior notion of what an algorithmic procedure is. In one sense, the Church-Turing thesis can be seen as offering a candidate formal definition for what an algorithmic procedure is — anything that can be done on a Turing Machine. And so far the definition has proved to be a good one: all things that have been studied that meet our vague intuitive notions of what it is to be algorithmic have turned out to be implementable on a Turing Machine.

11.1.1 Decision Problems & Effective Procedures

Let us, however, begin at the beginning by stating what a decision problem and an effective procedure are:

Decision Problem

A decision problem is a set of questions, where the answer to each question is either "yes" or "no".

Decidable Problem

A decidable problem is a decision problem for which there is an *effective procedure* (or algorithm) for answering each question in a finite amount of time.

Undecidable Problem

An undecidable problem is a decision problem for which no effective procedure exists

Thus, we can give a clear definition of a decision problem if we can give a clear definition of an effective procedure (i.e. algorithm).

Unfortunately, this cannot be done. For a particular programming language (for example), we could state what algorithms are possible given the constructs of that language (e.g. while loops, conditional statements, etc). But we can't be sure that some clever language designer won't come up with an entirely new kind of language construct that extends the range of possible algorithms.

At best, then, we can state some characteristics that effective procedures must exhibit

Necessary Properties of an Effective Procedure

1. It must be complete
   — be able to produce a positive or negative answer for each question in the problem domain
2. It must be mechanistic
   — consist of a finite sequence of instructions that can be carried out without requiring insight, ingenuity or guesswork
3. It must be deterministic
   — always produces the same answer when presented with the same input

Turing machines clearly satisfy all of these properties, and the essence of the Church-Turing thesis is to suggest TMs as a model for all effective procedures

Church-Turing Thesis (for decidability)

There is an effective procedure for solving a decision problem iff there is a Turing machine accepting a recursive language that solves the problem.
(I.e. for any decidable problem, a Turing Machine can be constructed to return the answers.)

(In a later section, we will extend the thesis to cover computability: i.e. returning more than just yes/no answers). This thesis, though not provable, is well confirmed. Nothing has yet been found that plausibly counts as an effective procedure that cannot be coded up as a Turing Machine. In other words, all computer algorithms correspond at some basic level to the operations of a Turing Machine.

11.1.2 Termination & Loops

Note that the Church-Turing thesis states that there must be a TM defining a *recursive* language that solves the decision problem. This is an important qualification. Recall that a TM defining a recursive language is one that halts (accepts) or crashes (rejects) for any input, and never goes into a loop. By contrast, a TM defining a recursively enumerable language is one that either halts (accepts), crashes (rejects), or goes into a loop (rejects).

A decidable problem is defined as being one for which an answer is always returned in a *finite* amount of time. If a TM is capable of delivering a negative answer by going into an infinite loop, we cannot guarantee that it always returns an answer in a finite amount of time. Hence, we cannot afford to solve a decision problem by means of a potentially loopy TM defining a recursively enumerable language.

It is worth dwelling on this point a bit more. A natural reaction is to say "OK, so a negative answer is given by going off into a loop. Can't we just detect when the machine has gone into a loop, and cease its execution there and then, returning a negative answer in finite time?"

The answer to this is "no, we can't". We will establish this more carefully in a moment by means of the halting problem. But the following argument shows why one initially plausible method of loop detection just won't work.

Suppose that we cannot observe the internal workings of the TM. Instead, we just provide it with its input, set it going and wait for an answer. This is much the same as running a C++ program without the aid of a debugger: you type in the command name and its arguments, hit "return", and hope. And as with C++ programs, it is not entirely unknown for the TM to go off into a loop.

The trouble is, from the outside there is no easy way to distinguish between when the TM is just taking a long time to compute an answer, and when it has really gone into a loop. One might decide to wait for a long time (several hours, maybe several thousand years), and if no answer has been returned by then hit Ctl-C and terminate execution. But at the precise point at which execution

is terminated, it could be that the machine was just a few steps away from returning the answer. If you had waited just a little longer, you would have discovered that the machine was not in a loop after all. And this observation applies no matter how long you wait before giving up and terminating execution. For after all, a finite wait for an answer can still be a very long wait.

We cannot detect loops, therefore, just by setting finite limits on the amount of time allowed for execution. For the limit will cut off some permissible finite executions. It therefore looks as though you have to look behind the scenes to determine if a TM is looping. But as we will now see, even this does not help.

11.2 The Halting Problem

The halting problem is as follows: given any Turing machine and input, will the machine halt on that input — yes or no? The halting problem is undecidable.

The undecidability of the halting problem does not mean that you can never detect loops in the execution of TMs. What it does mean is that you cannot detect *all* loops — some TMs on some inputs will have undetectable loops.[1]

The proof that the halting problem is undecidable is arguably one of the landmarks of 20th century thought. While the proof involves a certain amount of intuition stretching, it is basically quite simple. In outline, it goes as follows

1. First show how any Turing Machine can be encoded as a sequence of symbols that can be written on the input tape to a TM.

   (This should not be that surprising: if you look at the postscript file for these notes, you will see that all the TMs we have discussed have been represented as linear sequences of binary characters.)

2. Then suppose that we have succeeded in designing some TM, H, that solves the halting problem.

   That is we give the encoding of some TM, M, and its proposed input, I, as the input to H. We then run H on $\langle M, I\rangle$. H behaves as follows

   - If M halts on I, then H halts on $\langle M, I\rangle$ in an accepting state.
   - If M loops on I, then H crashes on $\langle M, I\rangle$ (halts in a rejecting state).

   We now deduce a contradiction from the supposed existence of a machine like H:

3. Given H, it is easy to modify it to form another machine H' that behaves as follows:

   - If M halts on I, then H' loops on $\langle M, I\rangle$

[1] In fact the halting problem is semi-decidable: we can return positive answers in finite time (e.g run the machine on the input and see if it halts), but we cannot return all negative answers in finite time.

- If M loops on I, then H' halts on $\langle M, I\rangle$

4. From H' we can construct another machine H'' that computes what happens when a TM is run on its own encoding.

   H'' takes as input the encoding of some TM, M, duplicates it, and then runs H' on the pair $\langle M, M\rangle$.

   The machine H'' behaves as follows

   - If M halts on its own encoding, then H'' loops on M's encoding
   - If M loops on its own encoding, then H'' halts on M's encoding

5. What happens if H'' is run on its own encoding?

   - If H'' halts on its own encoding, then H'' loops on its own encoding
   - If H'' loops on its own encoding, then H'' halts on its own encoding

   In other words, we get a contradiction.

   This means that a TM like H'' cannot exist.

   Since H'' was constructed from H by perfectly legitimate steps, it follows that a TM like H cannot exist.

The details of the proof are not that interesting, and consist in (a) showing how you can encode a TM on an input tape, (b) showing how you can construct H' from H, and (c) how you can construct H'' from H'. Nonetheless, we will briefly go over them.

11.2.1 Encoding TMs

To encode a TM, we can assume (without loss of generality) that all the TMs to be considered will have an input vocabulary $\{0, 1\}$, a tape vocabulary $\{0, 1, \Delta\}$ and states $\{s_0, s_1, \ldots, s_n\}$.[2] We can encode these elements in terms of the halting machine, H's, input vocabulary as follows

Symbol	Encoding	Symbol	Encoding
0	1	s_0	1
1	11	s_1	11
Δ	111	$\vdots$	$\vdots$
L	1	s_i	1^{i+1}
R	11		

Let $en(x)$ be the encoding of a symbol x.

We can encode a transition $\delta(s_i, x) = \langle s_j, y, D\rangle$ as

$$en(s_i)0en(x)0en(s_j)0en(y)0en(D)$$

[2]There is no loss of generality: after all, everything in a computer is represented as a binary string.

We can encode all the transitions on a tape by separating the individual transition codings by 00. Since a TM is completely defined by its transitions, this gives an encoding of the machine.

We can separate the machine encoding from its input by 000. Hence we can write on the tape of the halting machine H the encoding of the subject machine and its input.

11.2.2 Loops and Duplication

The proof that the halting problem is undecidable starts by assuming the existence of a halting machine H, such that for any Turing machine M and input I we have

Behaviour of H:

- If M halts on I, then H halts & accepts $en(M)I$
- If M loops on u, then H crashes & rejects $en(M)I$

From this we create a machine H' with the behaviour

Behaviour of H':

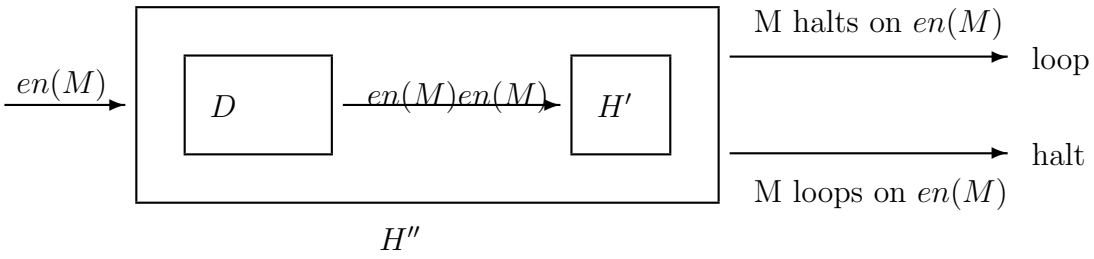
- If M halts on I, then H' loops $en(M)I$
- If M loops on u, then H' crashes on $en(M)I$

This is acheived quite simply if we assume that H is a TM with accepting states. To each accepting state, s_a, we add looping transitions:

$$
\begin{array}{l}
s_a \;\Rightarrow\; 1/1\,R \;\Rightarrow s_a \\
s_a \;\Rightarrow\; 0/0\,R \;\Rightarrow s_a \\
s_a \;\Rightarrow\; \Delta/\Delta\,R \;\Rightarrow s_a
\end{array}
$$

These just loop around on state s_a, whatever symbol is being scanned on the tape, and move the tape head towards the right. Since the tape extends infinitely far to the right, this guarantees an infinite loop.

Given the construction of H' from H, we now need to show how you can construct H'' from H'. We have already shown (p. 170) how to construct a Turing machine that duplicates whatever is on its input tape. Let us call this machine D. We construct H'' by first feeding a machine encoding through D to duplicate it on the input tape. We then pass control over to the machine H', whose input is the duplicated tape



The behaviour of this machine is

Behaviour of H''

- If M halts on $en(M)$, then H'' loops on $en(M)$
- If M loops on $en(M)$, then H'' halts on $en(M)$

From which we can obtain a contradiction by running H'' on $en(H'')$, and thus show that there can be no such machine as H.

11.3 Undecidable Problems

The unsolvability of the halting problem has a number of consequences. One of the more immediate ones is that it is impossible to write a software utility capable of detecting all loops in all possible programs. In other words, there is no general way of guaranteeing that an algorithm will always terminate — it is possible to spot some non-termination conditions, but not all of them.

But the halting problem has a much wider range of application. There are a whole variety of apparently unrelated decision problems that can be shown to be undecidable by *reducing* them to the halting problem.

Reducing a decision problem, P, to the halting problem means the following:

1. Assume that there *is* some effective procedure for solving P
2. Show that such a solution to P could be adapted to solve the halting problem
3. Since the halting problem is known to be undecidable, there cannot in fact be an effective procedure solving P.

In this section we will reduce a number of decision problems to the halting problem. In doing this, we will establish some undecidable properties of context free grammars, namely

- The problem of whether two context free grammars generate disjoint languages is undecidable

- The problem of whether a context free grammar gernerates all possible strings as sentences of its language is undecidable
- The problem of whether a context free grammar is ambiguous is undecidable.

To establish these results, we first look at semi-Thue systems and then the Post-correspondence problem.

11.3.1 Semi-Thue Systems

A semi-Thue system is an unrestricted rewrite grammar that has no start symbol and does not distinguish between terminal and non-terminal symbols. The rules are just used to rewrite one string into another.

For arbitrary strings u and v, the problem of whether $u \stackrel{*}{\Rightarrow} v$ is a semi-Thue system is undecidable.

This can be proved via reduction to the halting problem. From any determinstic TM we can produce an equivalent semi-Thue system (in roughly the same way that unrestricted grammars can be generated from TMs). It can be shown that $u \stackrel{*}{\Rightarrow} v$ in the derived semi-Thue system iff the computation in the TM will halt in an accepting state given input u.

But by the halting problem, there is no way of deciding whether the TM will halt. Consequently there is no decision procedure for $u \stackrel{*}{\Rightarrow} v$

11.3.2 The Post Correspondence Problem

Suppose that you have dominoes with strings over a given vocabulary written on the top and bottom halves. The post correspondence problem it to find a way of putting the dominoes together, so that the string taken from concatenating all the top halves is identical to that taken from concatenating the bottom halves, e.g.

aaa	baa	aaa
aa	abaaa	aa

Formally, a Post correspondence system consists of a vocabulary V and a set of pairs of strings $\langle u_i, v_i \rangle$ where $u_i, v_i \in V^+$ A solution is a sequence $i_1, i_2, \ldots i_k$ where

$$u_{i_1} u_{i_2} \ldots u_{i_k} = v_{i_1} v_{i_2} \ldots v_{i_k}$$

Assume that $V = \{0, 1\}$. Let S be a semi-Thue system over V. For a pair of strings u, v over V^* we can show that the correspondence system has a solution

iff $u \stackrel{*}{\Rightarrow} v$ in the semi-Thue system (for details see Sudkamp). Since the semi-Thue system is undecidable, so is the correspondence system.

11.3.3 (Un)Decidability Results for CFGs

Using the Post correspondence result we can establish some undecidability results for CFGs. From a Post correspondence system,

$$C = \langle V, \{\langle u_1, v_1\rangle, \ldots \langle u_n, v_n\rangle\}\rangle$$

we can construct two grammars (for the top and bottom halves of the dominoes)

G_t $N_t = \{S_t\}$
$V_t = V \cup \{1, 2, \ldots, n\}$
$R_t = S_t \rightarrow u_i S_t i \mid u_i i$ for $i = 1$ to n

G_b $N_b = \{S_b\}$
$V_b = V \cup \{1, 2, \ldots, n\}$
$R_b = S_b \rightarrow v_i S_b i \mid v_i i$ for $i = 1$ to n

If the correspondence system has a solution $i_1 \ldots i_k$ then the two grammars can derive the same string:

$S_t \stackrel{*}{\Rightarrow} u_{i_1} \ldots u_{i_k} i_k \ldots i_1$
$S_b \stackrel{*}{\Rightarrow} v_{i_1} \ldots v_{i_k} i_k \ldots i_1$
where $u_{i_1} \ldots u_{i_k} i_k \ldots i_1 = v_{i_1} \ldots v_{i_k} i_k \ldots i_1$

Theorem 1: Whether the languages of two CFGs is disjoint is undecidable.

Proof 1: Assume there is a decision procedure. Then we could solve the Post correspondence system:

- For an arbitrary correspondence system construct two grammars
- Use the decision procedure to determine whether the languages of the grammars are disjoint
- The correspondence problem has a solution iff the two grammars are non-disjoint

Theorem 2: There is no decision procedure for whether the language of a CFG is all possible string over the vocabulary

Theorem 3: There is no decision procedure for whether a CFG is ambiguous

Proof 3: Derive two CFGs from an arbitrary correspondence system, and combine them with the rules $S \rightarrow S_t \mid S_b$

There will be two derivations for some string iff the correspondence system has a solution.

11.3.4 Decidability Results for CFGs

Forunately, there are also some decidable questions about CFGs. Most of the proofs involve presenting an algorithm, but here we will just state the results.

1. There is a procedure to determine whether or not a CFG can generate any sentences
2. There is a procedure to determine whether or not a given non-terminal is ever used in the generation of a sentence

   (See detection of useless symbols in connection with Chomsky Normal Form)
3. There is a procedure to determine whether a CFG generates a finite or an infinite language.

   (Basically, if it generates any sentences long enough to apply the pumping lemma, it generates an infinite language)
4. There is a procedure to determine whether a given string is a sentence of the language defined by a given CFG

11.3.5 Another Undecidable Problem

We will finish with one further undecidable problem, namely whether one proposition entails another in first-order predicate calculus.

It is possible to encode any Turing machine as a conjunction of propositions in the predicate calculus. We can put together a conjunction of formulas Γ that gives a logical encoding of the machine and its input. We can also construct a formula H stating that the machine halts. Putting these together, we get "Γ entails H", which means that the machine does halt for a given input. But since we already know that this is an undecidable problem, the entailment between Γ and H must be undecidable.

In fact, the problem is *semi-decidable*. This means that for any pair of propositions, ϕ and ψ, if ϕ entails ψ, then there is a decision procedure that will inform us of this in a finite time. However, if ϕ does not entail ψ, we cannot always be informed of this fact in a finite amount of time.

A semi-decidable problem is one for which positive (or negative) answers are always given in a finite amount of time, but where negative (or positive) results may cause a loop.

11.4 Computability

In talking about decidability, we have just been concerned about whether or not a TM halts, regardless of what is written on the tape when the TM halts. But algorithms/TMs can clearly do much more than decide yes/no answers.

They can compute more interesting results, and in the case of TMs leave these results written on the tape.

There is another version of the Church-Turing thesis for computability

Church-Turing Thesis (Computability)
Any computable function can be computed on a Turing machine

As with the Church-Turing thesis for the decidability the suggestion is to take Turing machines as definitional for algorithms, and subsequent investigation has so far shown this to be reasonable.

In this section we will (i) look at a particular form of TM used for computing functions, (ii) show that some some arithmetic functions are inherently uncomputable, and (iii) very briefly relate Turing computability to the theory of recursive functions.

11.4.1 Turing Computability

A Turing machine that computes a function has two distinguished states: a start state s_0 and a single accepting state s_f. The input to the function is written in the standard position on the tape. There is one transition out of s_0, which moves the tape head to position 1, and no transitions back into s_0. All computations that terminate do so in state s_f. On termination, the result of the function is written in the standard position on an otherwise blank tape.

A function is Turing computable if there is a TM of the above form that computes it.

11.4.2 Uncomputable Functions

Some functions are not Turing computable. Essentially, each TM defines a (computable) function. However there are more functions on real numbers than there are possible Turing machines, and so some numeric functions are therefore uncomputable.

In Chapter 2 we showed that the set of real numbers was uncountably large, via Cantor's diagonalisation argument.

Turing machines are defined by their transitions, which are 5-tuples where each element is taken from a countable set. The set of all possible 5-tuples is therefore countable (though infinite).

Hence there are more real numbers than possible Turing machines, even though there are an infinite number of both.

Consider all unary numeric functions applied to a given input: there are as many outputs (and hence as many possible functions) as there are real numbers — uncountably many. But each Turing machine can only return one result, and there are only countably many of these. Thus some unary functions get left

out. (Whether there are any interesting or important uncomputable functions is another question).

11.4.3 Recursive Functions

An alternative approach to defining computability has been in terms of recursive function theory (covered in more detail in G53COM). A very large family of functions can be built up from three basic primitive recursive functions

1. Successor: $s(x) = x + 1$
2. Zero: $z(x) = 0$
3. Projection: $p_i^n(x1, \ldots, x_n) = x_i$, $1 \leq i \leq n$

via the following modes of combination

1. Composition

   Let $g_1, \ldots, g_n$ be k-place functions and h be an n-place function.

   Composition of h with $g_1, \ldots, g_n$ is a k-place function f

   $$f(x_1, \ldots, x_k) = h(g_1(x_1, \ldots, x_k), \ldots, g_n(x_1, \ldots, x_k))$$

   (That is, the function obtained by using the output of $g_1, \ldots, g_n$ as the input to h)

2. Primitive Recursion

   Let g be an n-place function and h an $(n+2)$-place function.

   We can obtain an $(n+1)$-place function, f, from them by primitive recursion

   (a) $f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$ (base case)
   (b) $f(x_1, \ldots, x_n, y+1) = h(x_1, \ldots, x_n,\ y,\ f(x_1, \ldots, x_n, y))$ (recursion)

Functions built up from Zero, Successor and Projection via compsition and primitive recursion are known as primitive recursive functions. This encompasses a surprisingly large range, for example

Some Primitive Recursive Functions

Description	Function/ Predicate	Definition
addition	$add(x,y)$	$add(x,0) = p_1^2(x,0) = x$
	$x+y$	$add(x,y+1) = s(add(x,y))$
multiplication	$mult(x,y)$	$mult(x,0) = z(x) = 0$
	$x.y$	$mult(x,y+1) = mult(x,y) + x$
predecessor	$pred(y)$	$pred(0) = 0$
		$pred(y+1) = y$
proper subtraction	$sub(x,y)$ $x \dot{-} y$	$sub(x,0) = p_1^2(x,0) = x$ $sub(x,y+1) = pred(sub(x,y))$
exponentiation	$exp(x,y)$ x^y	$exp(x,0) = s(z(x)) = 1$ $exp(x,y+1) = exp(x,y).x$
sign	$sg(x)$	$sg(0) = z(0) = 0$
		$sg(y+1) = s(z(0)) = 1$
sign complement	$cosg(x)$	$cosg(0) = 1$
		$cosg(y+1) = 0$
less than	$lt(x,y)$	$sg(y \dot{-} x)$
greater than	$gt(x,y)$	$sg(x \dot{-} y)$
equals	$eq(x,y)$	$cosg(lt(x,y) + gt(x,y)$

If one adds one further mode of function combination, known as *minimalisation* one obtains the set of μ-recursive functions:

3. Minimalisation Given a predicate p (i.e. a function that returns 1 or 0), the minimalisation
$$\mu z[p(x_1, \ldots, x_n, z)]$$
gives the least value of z for which $p(x_1, \ldots, x_n, z) = 1$

Investigation has shown μ-recursive functions to be a good way of characterising what we intuitively regard as computable functions.

However, it turns out to be straightforward to convert any μ-recursive function into a Turing Machine, so the Turing Computabilty gives the same characterisation of computability as recursive function theory.

Conversion of the building blocks of primitive recursive functions to TMs proceeds, roughly, as follows:

- The successor TM just adds an extra 1 to the end of the number (represented as a string of 1's) on the input tape.
- The zero function deletes all but the first 1.
- Projection functions scan along the tape until they find the number they need to return, copy the number to the front of the tape, and delete everything else.

- Composition involves putting two or more TMs together so that the output that one writes to the tape is used as input to the next.
- We can also combine TMs for two functions in such a way as to compute the primitive recursion of the two functions. (For details, see Sudkamp).
- Similarly for minimalisation

11.5 Summary

This chapter has been about the Church-Turing thesis, which amounts to saying that anything that can be algorithmically computed can be computed via a Turing Machine. The halting problem shows that there are some well defined and interesting problems to which algorithms cannot always return a yes-or-no answer in finite amount of time. Unfortunately, a large number of problems are undecidable in this sense, as can be shown by reducing them to the halting problem. Finally, we showed that some functions cannot be computed (decidably or otherwise) by Turing Machines. Since Turing Machines give an abstract characterisation of the expressive power of digital computers, this means there are theoretical limits on what can be done with computers.

THE END